LOTTERY SCHEDULING IN THE LINUX KERNEL: A CLOSER LOOK

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

David Zepp

June 2012

COMMITTEE MEMBERSHIP

TITLE: LOTTERY SCHEDULING IN THE LINUX KERNEL: A CLOSER LOOK

AUTHOR: David Zepp

DATE SUBMITTED: June 2012

COMMITTEE CHAIR: Michael Haungs, Associate Professor of Computer Science

COMMITTEE MEMBER: John Bellardo, Assistant Professor of Computer Science

COMMITTEE MEMBER: Aaron Keen, Associate Professor of Computer Science

ABSTRACT

LOTTERY SCHEDULING IN THE LINUX KERNEL: A CLOSER LOOK

David Zepp

This paper presents an implementation of a lottery scheduler, presented from design through debugging to performance testing. Desirable characteristics of a general purpose scheduler include low overhead, good overall system performance for a variety of process types, and fair scheduling behavior. Testing is performed, along with an analysis of the results measuring the lottery scheduler against these characteristics. Lottery scheduling is found to provide better than average control over the relative execution rates of processes. The results show that lottery scheduling functions as a good mechanism for sharing the CPU fairly between users that are competing for the resource. While the lottery scheduler proves to have several interesting properties, overall system performance suffers and does not compare favorably with the balanced performance afforded by the standard Linux kernel's scheduler.

Keywords: lottery scheduling, schedulers, Linux

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

1. Introduction

A scheduler is the piece of an operating system that allocates CPU time to individual processes. An operating system can appear to run multiple processes simultaneously by sharing the CPU; giving each process a bit of CPU time. The scheduler allows a process to run for a period of time, and then chooses another process and allows it to run.

The scheduler is responsible for scheduling all processes. Processes can broadly be classified into three types; CPU bound, I/O bound, and real-time. I/O bound processes block often for I/O and use the CPU only in short bursts. Interactive programs are generally I/O bound as they are waiting on the user for input. For an interactive workload, timing and latency are important in order to support the desired user perception of sole possession of the machine. For the CPU bound process, processor access is not as timing critical. For a CPU bound process the focus is making progress on and finishing a fixed amount of work as quickly as possible. Real-time processes are the most timing critical as the computer must be able to respond within a set amount of time in order to meet hard deadlines. This research explores both I/O bound processes and CPU bound process performance as well as the balance of I/O and CPU bound processes. Real-time processes are not considered.

Another goal of scheduling is fairness; defined as each user getting an equal share of the CPU over the long run. As part of the fairness dilemma, the scheduler must balance users owning few processes with users owning many. While fairness is

certainly a desirable element in some environments, not all schedulers consider fairness to be a goal.

"Lottery scheduling" is a mechanism that can be used for allocation of CPU time. This paradigm was introduced by Carl Waldspurger and William Weihl (1994) in "Lottery Scheduling: Flexible Proportional-Share Resource Management". As part of their research Waldspurger and Weihl implemented lottery scheduling in the Mach kernel. This paper is the basis of all lottery scheduling and related variants. The paper has had a far reaching and lasting impact, and nearly all the works cited in the later review of related works reference Waldspurger and Weihl.

This research explores how the Linux OS performs for a set of tests using the lottery method of scheduling as compared to both the version 2.4 kernel's native priority based scheduler, and a version 2.4 kernel patch that implements "fair-share" scheduling. Through implementation and testing of the scheduler, this novel idea proposed by Waldspurger and Weihl (1994) is investigated. This work explores the purported CPU resource allocation strengths of the lottery scheduler, as well as overall performance across the CPU bound and interactive load profiles. An in depth analysis of the order of a process' lottery wins, and the impact this ordering can have in the short term is performed. Kernel level profiling is presented that demonstrates the functioning of the lottery. In addition, an analysis of the classic compensation ticket function identifies shortcomings that reinforce results published by others (Petrou, Milford, & Gibson, 1999). Finally, the results of a practical application experiment that uses lottery scheduling to provide different service levels for web servers running on a single system are presented.

The next chapter of the paper gives some background on the schedulers that are tested and contrasted for this research. In Chapter 3 the Lottery scheduling API is introduced and described. The design and implementation of the lottery scheduler is described in Chapter 4. Chapter 5 details the experiments performed and presents analyses of the results. A review of current works and the conclusions drawn from the research are presented in chapters 6 and 7 respectively. Finally, Chapter 8 suggests several topics to pursue as future work.

2. Scheduling Background

A multi-tasking operating system provides the illusion of executing multiple processes simultaneously. It is able to create this illusion by running multiple programs in a serial fashion for small periods of time. Below are descriptions of the schedulers that are compared in this research, followed by a detailed discussion of the lottery scheduler and its implementation for this project.

2.1 Description of Default Linux 2.4 Kernel Scheduler

The 2.4 version Linux kernel's scheduler is a priority based scheduler with a dynamic component to its priority calculation. The scheduler selects processes by priority, and adjusts priority to ensure that favorable system performance is achieved. From a scheduling perspective, processes can be divided into two broad classes. I/O bound processes that spend much of their time waiting for I/O, and CPU bound processes that use all the CPU time they are allocated. Excluding the requirements of real-time processes, the goal of a scheduler is to keep CPU bound processes running as much as possible while servicing I/O bound processes in as timely a fashion as possible. With this goal in mind, the Linux scheduler assigns higher priorities to processes that are I/O bound, while lowering the priority of CPU bound processes. This action is the dynamic component of the priority based scheduler.

Processes are given an amount of time measured in clock ticks from the Programmable Interval Timer (PIT). The processes are then run until their CPU time allocations are consumed. Then a new allocation to all processes is performed and the cycle begins again. This complete cycle of process execution and allocation of

CPU time is referred to as an epoch. The amount of time allocated to a process within an epoch is a function of the process's static priority and the amount of CPU time that the process used in the prior epoch.

During an epoch processes are executed in the order defined by their priorities. The runqueue is a single queue in which all runnable processes are stored. The scheduler loops through the queue and selects the process with the highest priority to run next. This assessment of priority is calculated in the *goodness*() function. When all runnable processes have completely consumed their time quantums the current epoch is over and a new epoch begins.

At the beginning of the new epoch a new time quantum is calculated for all processes. A process' new time quantum is calculated from the combination of the process' static priority and the number of clock ticks remaining from the prior epoch. In this way a process that does not use its entire time quantum in an epoch is given a boost in priority. The measurement of priority calculated in the goodness() function is based in large part on the size of a process' time quantum.

2.2 Description of Rik van Riel's FairSched Kernel Patch

"With this patch your system will not allow any user to monopolise the CPU by firing up a lot of CPU hogs"; this is Rik van Riel's description of the FairSched kernel patch, available at http://www.surriel.com/patches/2.4/2.4.19-fairsched. This is a patch to the 2.4.19 version kernel that introduces an element of fair-share scheduling. Fair-share scheduling focuses on providing equal allocation of CPU to each user on a multi-user system. This patch tracks CPU usage by user. The patch does its work during the end-of-epoch recalculation where each process is given a

new time quantum. If a user has tasks that use small or no amount of CPU time, they are given a small boost in priority through a larger than normal time quantum. If a user has a single task that is CPU bound, it will receive a standard time quantum, and will not be penalized. If a user has more than one task that is CPU bound some of the user's tasks will receive time slices that are smaller than the normal time quantum. In an extreme case, some of the processes may receive no time quantum for some number of epochs. The algorithm rotates processes in the task queue such that all processes of even the busiest user, will eventually receive some amount (however small) of time quantum. This rotation avoids the problem of starvation. This paradigm is similar to lottery scheduling in that resource rights of all users are preserved. Similar to my implementation of lottery scheduling, the root user is given no "free pass", and must adhere to the same rules as all other users. The "FairSched" kernel is different than the lottery scheduling function in that the user has no control over the way the de-prioritized processes are penalized. With lottery scheduling a user with multiple CPU hogs can specify which processes should still take priority by setting relative ticket ratios between processes.

2.3 Description of the Lottery Scheduler

The Lottery scheduling function works by giving each process a set of numbered lottery tickets. When the scheduler is ready to grant the CPU to a process it holds a "lottery", picks a random winning ticket, and the process with the winning ticket is given the CPU. A process' odds of receiving the CPU are directly related to the number of tickets that the process holds. Lottery scheduling offers some features that are not available with standard priority based schedulers. One such feature is

accurate control over relative process execution rates. With this control a process can be run exactly three times faster than another, or 20 times faster. Other schedulers have mechanisms that allow processes to be given different priorities, but none surveyed in this research offer the degree of accuracy to control relative rate of execution that is provided by lottery scheduling. Another feature of lottery scheduling is the capability to isolate scheduling policy within trust boundaries. This sort of policy is not easily enforceable with many conventional schedulers. User accounts are a trust boundary where isolation occurs in this project's implementation of lottery scheduling.

3. Lottery Scheduling API and User Interface

The API consists of both system calls and command line utilities that can be used to monitor and manipulate both base ticket and process ticket counts.

3.1 System Calls

The system call interface to the lottery scheduler allows for manipulation of process tickets and base tickets (the two types of lottery tickets in this implementation). Process ticket count can be set for a process with the *set_process_tkts* system call. Process ticket count for a process can be obtained with the *get_process_tkts* call. In a similar fashion base tickets for a user can be set with *set_base_tkts* and the current level can be obtained with *get_base_tkts*. The "get" system calls allow any user to get the base or process ticket counts for any other user or process on the system. Two other system calls were added to assist in debugging and profiling the lottery scheduler. The *get_lott_wins* and *get_counter_info* system calls can be used to monitor lottery win and time slice usages metrics.

3.1.1 Setting Process Ticket Count - set_process_tkts (pid_t pid, int processtkts)

The *set_process_tkts* system call takes a pid and a positive integer count of process tickets as arguments. The valid range of process tickets is an integer from 1 to 10,000. This range restriction is meant to provide a reasonable range of ticket values. It may be interesting to experiment with allowing 0 as a way of suspending the execution of a process. The system call first verifies that the *pid* is a valid active process, returning an error if not. The system call also checks to see if the user

requesting the change owns the process or has the *CAP_SYS_NICE* capability (a Linux security mechanism). Generally only a process's owner (or the root user) can change the process ticket count. The process's ticket count is set to the count specified by the parameter *processtkts*. An additional update to a dynamically maintained kernel variable that holds the per-user count of process tickets for runnable processes is performed if the process is on the runqueue. The purpose of altering process ticket counts is to control relative rate of execution between a single user's processes. A process' ticket count may be altered with this system call at anytime, and there are no limitations with regards to the timing or frequency of these alterations.

3.1.2 Getting Process Ticket Count - get_process_tkts (pid_t pid, int *tkts)

The *get_process_tkts* system call takes a *pid* as an input parameter and an integer output parameter for the number of process tickets that the process specified by the *pid* current holds. After the *pid* is determined to be that of a process currently in existence, the number of process tickets allocated to the process is returned in *tkts*. If *pid* is not the PID of a existing process, an error is returned. There are not any restrictions on calling this function. Any process can be queried for ticket count by any user.

3.1.3 Setting Base Ticket Count - set_base_tkts (uid_t uid, int basetkts)

The *set_base_tkts* system call takes a *uid* and a count of base tickets as parameters. The valid range of base tickets is from 100 to 10,000. Much like the range defined for process tickets in the *set_process_tkts* call this range restriction is

simply a conservative estimate of a reasonable range. Here too it may be interesting to experiment with setting base tickets to 0 for a user as a way of freezing the user and all of the user's processes. The *set_base_tkts* system call first uses the specified *uid* to locate the *user_struct* in the *uidhash_table* array of user_structs. If a *user_struct* for the *uid* does not exist, then the system call returns an error. If the *user_struct* for the *uid* does exist, then the base tickets are altered contingent on the calling user's qualifications. The qualifications are similar to those required to manipulate *nice* values for a user. Only a user with the *CAP_SYS_NICE* capability can alter the base tickets of another user, or increase base tickets beyond the system default. Altering base ticket count controls the relative CPU resource rights between users.

An important point to make is that any changes made to base tickets are only in effect while the user's *user_struct* kernel structure exists in the kernel. This structure is created at the point in time that the first process belonging to a user is established. A *user_struct* structure is initially created with the default number of base tickets allocated to all users (DEF_BASE_TICKETS). This number of base tickets can be altered, but any altered quantity is semi-transient, as the *user_struct* structure is destroyed as soon as the last process belonging to the user is terminated. The following example illustrates this important subtlety.

1. *Process1* for *user A* is created.
2. Before the creation of this process *user A* had no other processes in existence, so a *user_struct* structure is created, and the base tickets field is populated with DEF_BASE_TICKETS.

3. The base ticket count for *user A* is doubled with the *set_base_tkts* system call.
4. *Process2* for *user A* is created. The *user_struct* structure for *user A* already exists; created when *Process1* was launched.
5. *Process2* runs and enjoys the benefit of the elevated base ticket status of its owner.
6. *Process1* and *Process2* terminate, and the *user_struct* structure for *user A* is destroyed.
7. *User A* starts *Process3*. Since the *user_struct* must again be created, it is done with DEF_BASE_TICKETS base tickets for *user A*.

Worth noting at this point, is that the Linux kernel is *supposed* to deallocate the *user_struct* when a user's last process is removed from the system. In my testing, I noticed that the base lottery tickets of a user remained fixed at the point at which they were last set by a call to *set_base_tkts*. This was true even after the user had logged off the system, and all of the user's processes had terminated. In fact, a bug prevented this deallocation from happening properly in the mainline kernel until after version 2.5.74. Because of this bug, the *__count* reference count variable of the *user_struct* never decrements further than 1, and the *user_struct* is never removed from the *uidhash_table* array of *user_structs*. This explained why I was observing this unusual behavior with base tickets levels that got "stuck". To further complicate the situation, the SUSE distribution (kernel version 2.4.12-99-default) that I used for some initial kernel exploration had fixed this bug. Of course the vanilla 2.4.31 kernel that I altered for the lottery scheduler still had the bug. It took many hours of

debugging to find this bug and understand why kernel versions 2.4.12-99-default, and 2.4.31 behaved differently.

3.1.4 Getting Base Ticket Count - get_base_tkts (uid_t uid, int *tkts)

The *get_base_tkts* system call takes a *uid* as an input parameter and an integer output parameter for the number of base tickets that the user specified by the *uid* currently holds. This system call first checks the uid against the kernel's list of currently active users; that is those users with at least one process running. If the uid is not found in this list an error is returned, otherwise the user's base ticket count is returned in *tkts*. There are not any restrictions on running *get_base_tkts*. It can be run at any time and may be run for any user account by any other user.

3.1.5 Counting Lottery Wins - get_lott_wins (pid_t pid, int *wins)

This function is used to retrieve the current number of lottery wins for a process into the output parameter *wins*. This can be used in analysis, for example, to compare the number of wins that two processes received over the same period of time. This can be called on any process by any user account. If the PID specified by *pid* is not valid, an error is returned.

3.1.6 Monitoring Time Slice Usage - get_counter_info (pid_t pid, int index, int *counter_val)

This is used to return statistics on the portion of time quantum (the *counter* kernel variable) that a process uses each time it runs on the CPU. The full time quantum is represented by 6 ticks of the clock (a *counter* value of 6). The get_counter_info system call takes two input parameters. One is a pid and the other is

an index value from 0 to 6. The call returns the number of times that the pid was left with the *index value* in the kernel *counter* variable. For example, if *index value* 4 is passed into the system call, the call returns the count of times that the process was left with 4 ticks of time quantum when it relinquished the CPU. The kernel *counter* variable is decremented as the process' time slice is consumed. This can be used to gauge how CPU bound or I/O bound a process is. A very I/O bound process will rarely use much of the time quantum, where a CPU bound process will use as much as it can before it is preempted.

3.2 Command Line Utilities

The system calls were used to create a set of utilities that a user may employ from the shell to monitor and control base and process tickets. There are four utilities that can be used to get and set tickets counts. The *lnice* utility is used to spawn a process at an altered process ticket count. The *lrenice* utility is used to alter the process ticket count of a running process. The *lsetbase* utility sets the user base ticket count for a named user. Note that the fundamental behavior of these utilities is significantly different. A change to a process's process ticket allocation is only effective for a single process, and only for as long as the process exists. However, a change to a user's base ticket allocation impacts all of the user's processes immediately, and is effective until changed again or until all of the user's processes are terminated and the *user_struct* kernel structure removed. The *ltix* utility is used to report the number of process tickets for a process or the number of user base tickets for a named user.

3.2.1 lnice

```
Usage
lnice [-p] [TICKET_COUNT] [COMMAND [ARGS]...]
Run COMMAND with an adjusted number of lottery tickets.
Option: -p [10-10000] Adjusts process tickets
Ticket Ranges are from 10 (lowest) to 10000 (highest)
Example lnice -p 500 ps
With no adjusting ticket quantity, the COMMAND is run at default ticket allocation
Example lnice ps
```

lnice spawns a process at the process ticket count specified by the –p argument. When a process is launched from the shell it will start at the default process ticket count which is defined by the *DEF_PROCESS_TICKETS* kernel symbol. This utility can be used to launch a process with a process ticket count that is different from the default amount.

3.2.2 lrenice

```
Usage
lrenice [PID] [TICKET_COUNT]
This utility will facilitate changing the
process ticket count of a running process
PID: pid
TICKET_COUNT: Range of valid values are from 10 (lowest) to 10000 (highest)
Example <lrenice 515 1000> will change process 515's process ticket count to 1000
```

lrenice is used to alter the process ticket count of a running process. Once altered the process ticket count will stay the same until altered again or until the

process terminates. The process does not need to be running in the sense of being on the runqueue, rather it simply needs to be a process that exists.

3.2.3 lsetbase

```
Usage
lsetbase [-u] [USER] [TICKET_COUNT]
Change base ticket count of a user.
Option: -u [user_name] [100-10000] Adjusts user's base tickets
Valid Base Ticket values are from 100 (lowest) to 10000 (highest)
Example lsetbase –u paul 500
```

lsetbase is used to alter the base ticket count for a user. The user must have at least one process active on the system. This utility calls the *set_base_tkts* system call described above and is thus restricted by the *CAP_SYS_NICE* capability requirement.

3.2.4 ltix

```
Usage
ltix [-u|-p] [ARG]
OPTION: -u  ARG: user
Display base ticket count for user specified in ARG.
Example: ltix -u paul will report the number of user base tickets for user paul
OPTION: -p  ARG: PID
Display process ticket count for PID specified in ARG.
Example: ltix -p 1235 will report the number of process tickets for pid 1235
```

ltix is used to report process tickets for a given process or base tickets for a given user. If either the PID or the user specified in the argument is invalid, an error is returned.

4. Lottery Scheduler Design and Implementation

Waldspurger's lottery mechanism (Waldspurger & Weihl, 1994) provides two fundamental features; responsive control over the relative rate of process execution and modular resource management. Relative rate of execution is managed through the allocation of lottery tickets. Modular resource management is the ability to create subsets of a client population and protect the complete population from the local resource management policies of each subset. These subsets often reflect trust boundaries and Waldspurger labeled the tickets available within each subset a currency. In Waldspurger's design, any set could be further subsetted, with the tickets of one currency backing those of another. The lottery mechanism can be used to manage resources of many types, however for this project it is used only to manage the CPU resource.

In this project's implementation of lottery scheduling, the Linux user account is the trust boundary at which modular resource management occurs, and a user's processes are the mutually trusted clients that receive resource rights (Figure 1). This provides an element of fairness as each user is able to institute their own local resource management policy with no impact on the resource rights of other users. The choice to limit trust boundaries to user accounts results in an implementation that is not as extensible as Waldspurger's design, but simpler to execute. Each user is assigned a quantity of *base* tickets. The root user receives these *base* tickets as part of the statically defined *root_user* structure. Other users receive these tickets when the first process owned by the user is started. This is executed in the *alloc_uid*

function. The *base* tickets that all users possess are of comparable value, and the lotteries held by the scheduler are done so in terms of *base* tickets.

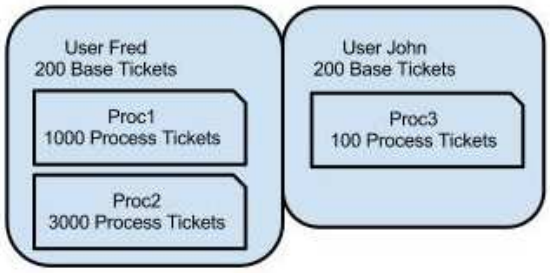

Figure 1 Base tickets control global CPU resource rights while process tickets are used to define local resource management policy. Equal base ticket allocations mean that users Fred and John have equal CPU resource rights regardless of process ticket allocation. Process ticket allocation regulating the execution rates of each process relative to others in the same domain, is policy specific to each user that has no effect on overall CPU resource rights.

Waldspurger's (Waldspurger & Weihl, 1994) notion of a client that receives resource rights equates to the Linux process. Each Linux process is assigned a default number of *process* tickets, in the currency of the process's owner. A user is free to assign any positive integer number of currency tickets to the user's processes (Figure 1). This freedom to alter ticket count within a currency is described by Waldspurger as ticket inflation/deflation. The *process* tickets are backed by the *base* tickets and any ticket inflation/deflation only impacts processes of a similar currency, that is to say it only impacts those processes owned by the user that represents a particular currency.

When the scheduling function is called, a lottery is held to determine the next process that is to be serviced by the CPU. In general terms, each runnable process'

process tickets are scaled into base tickets through a conversion calculation and a random number is selected in the range of 1 to *N* where *N* is the total number of calculated base tickets. The winning process is identified by matching the random number in the sequence of calculated based tickets logically overlaid onto the queue of runnable processes (Figure 2).

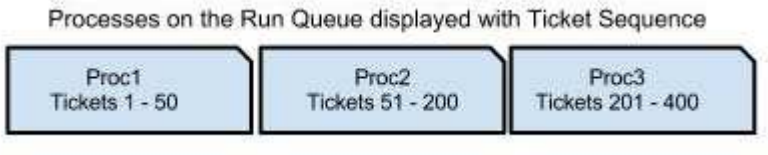

Figure 2 A winning process will be selected from this queue of runnable processes that have had their process tickets scaled into base tickets. The random number selected for this single lottery instance will be between 1 and 400. Only runnable processes are considered in each lottery.

4.1 Tickets and Ratios

To understand the lottery mechanism in greater detail, consider this example. For our example users John and Fred have both been assigned 200 base tickets. Each row in the table (Table 1) below represents a runnable process and all runnable processes on the system are listed in the table. Other processes may exist, but only the runnable ones are considered in a lottery. The first column lists the user that owns the process. The second column lists the runnable process; in this example the processes have been given different names for clarity. The third column titled "Process Tickets" lists the number of tickets that have been granted to each process by the process owner. This can be any positive integer value. The fourth column "Base Ticket Value" represents the process's "Process Ticket" count in terms of base

tickets. The last column maintains a running total of base tickets. This running total is used when determining the winner of the lottery.

Table 1 Table used in description of the lottery scheduling algorithm's action.

User	Process	Process Tickets	Base Ticket Value	Running Base Ticket Count
Fred	Proc1	1000	50	50
Fred	Proc2	3000	150	200
John	Proc3	100	200	400

Note that user Fred has two runnable processes, and that process tickets have been assigned by Fred such that a 3:1 ratio is achieved between Proc2 and Proc1. This ratio is reflected when scaling process tickets into base tickets. In the conversion to base tickets Fred's 200 base tickets are allocated to the two processes with a 3:1 ratio. User John has only one runnable process, so all of his base tickets are allocated to his single runnable process. The conversion to *base ticket value* is done by v=b*p/t where v is *base ticket value*, p is the *process* tickets assigned to the process, t is the total number of process tickets a user has allocated for runnable processes, and b is the user's base ticket count. For Proc1 *base ticket value* is 1000/4000*200 = 50.

Once all runnable processes' process tickets are converted to b*ase ticket value*, the lottery to select a winner is held. In the example there are a total of 400 "active" base tickets, so a number from 1 to 400 must be randomly selected. The total number of "active" base tickets is the sum of base tickets belonging to users who have a least a single runnable process. In this case only Fred and John have runnable processes. The winner of the lottery is determined by locating the winning ticket's position in

the *running base ticket count*. For example, if the winning ticket is number 201, user John is the winner as his tickets span the range from 201 through 400.

In summary, *process* tickets are used to control resource rights within a single user's sphere of influence, while *base* tickets represent the user's resource rights relative to all other competing users.

4.2 Compensation Tickets

In Waldspurger's Lottery Scheduling paper (Waldspurger & Weihl, 1994), he introduces a mechanism he calls compensation tickets that is designed to compensate processes that do not use all of their allotted time on the processor. Yielding the processor before a process's time slice is exhausted, is a characteristic of I/O bound processes. This compensation action is accomplished with an additional allocation of tickets to a singular process. The additional tickets are allocated based on the fraction of the time slice that was used. The goal of these extra tickets is to increase the process's chances in the next lottery, thereby ensuring that a compensated process is able to restart quickly when it is runnable. The additional allocation of compensation tickets is performed in a *base* ticket denomination rather than with *process* tickets. The additional tickets are associated with a single process only, and can not be transferred to another process or user. Any compensation tickets held by a process are revoked after the process wins a lottery; that is to say compensation tickets do not accrue. Compensation tickets are calculated in the *schedule()* function before the lottery drawing is performed. When the schedule function occurs, compensation tickets are calculated for the process that ran prior to the call to *schedule()*. In this explanation, this process will be referred to as the *previous* process; in code it is

actually referred to as *prev*. The *counter* variable attached to every process is used to calculate this fraction of time slice used. The *counter* variable is set to a specified value when a process starts it time slice. At every tick of the system clock, the counter variable is decremented by 1. When *counter* reaches 0 the time slice has been fully consumed. The complete compensation ticket calculation is (v*t/u)-v where v is *base ticket value* as described earlier, u is the portion of the time slice that was used, and t is the length of the time slice.

In order to visualize how compensation tickets are allocated and their impact on a lottery, we will extend our prior example to include them (Table 2). Suppose that Proc1 just ran, but yielded the CPU almost immediately after starting. The value in Proc1's counter variable is 5, signifying that it used only 1 clock tick of its total time slice (6 ticks). The compensation ticket calculation is *base ticket value * total time slice* / *used time slice – base ticket value*; or 50*6/1-50=250 compensation tickets.

Table 2 Table used in description of compensation ticket calculation.

User	Process	Process Tickets	Base Ticket Value	Compensation Tickets	Running Base Ticket Count
Fred	Proc1	1000	50	250	300
Fred	Proc2	3000	150	0	450
John	Proc3	100	200	0	650

These additional compensation tickets change the running base ticket count, and when the lottery is held this must be accounted for. In the example, with the addition of compensation tickets, the random number that is selected will be in the range of 1

to 650. Notice in the example, the compensation ticket allocation represents a significant boost in Proc1's chances to win a lottery; from 1 in 8 to 6 in 13.

Below is a detailed discussion of the implementation of compensation ticket calculation. First a look at the entire code block (Figure 3).

```
        if(prev == idle_task(this_cpu) || prev->policy != SCHED_OTHER) {
        }
        else  {

          /* Track statistics on how much of timeslice was used. */
          prev->counter_vals[prev->counter]++;

        /* if entire slice wasn't used */
          if(prev->counter > 0) {

            /* If task is still on runqueue then prev->process_tkts are already
               included in user_tpt */
            /* The task could still be on the runqueue if its state is still runnable... */
            if(task_on_runqueue(prev)){
              tmp_tpt = prev->user->user_tpt;
            }
            else { /* This is the case where the task got dequeued from the runqueue above */
                   /* Since it got dequeued, the user_tpt value was decremented, so we need
                      to add the previous processes process tkts back into the user total
                      process tickets */
              tmp_tpt = prev->user->user_tpt + prev->process_tkts;
            }

            /* tmp_base_cnt is the process's process tix scaled into base tix  */
              tmp_base_cnt = (prev->process_tkts*prev->user->user_base_tickets)/tmp_tpt;
            /* be sure that in cases where user_base_tickets is < than tmp_tpt
               we get at least one ticket for the process */
            if (tmp_base_cnt < 1) {
                tmp_base_cnt = 1;
            }

            /* In this case not even a single full tick was used */
            if( prev->counter == LOTTERY_SLICE ){
              /* Usage of constant 59 explained in thesis document */
              prev->process_comp_tkts = tmp_base_cnt*59;
            }
            else {
              prev->process_comp_tkts = ((tmp_base_cnt*LOTTERY_SLICE)/\
              (LOTTERY_SLICE - prev->counter)) -tmp_base_cnt;
            }

          } /* End of Standard Comp. Tix Calc */

#ifdef CONFIG_PB_LOTTERY
          pwr_raise = prev->counter;
          pwr_boost = 5;
          while (pwr_raise) {
            pwr_boost = pwr_boost*5;
            pwr_raise--;
          }
          prev->process_comp_tkts = prev->process_comp_tkts + pwr_boost;
#endif  /*  CONFIG_PB_LOTTERY */

          /* The prev task can still be on the runqueue, e.g. when it's preempted */
          if(task_on_runqueue(prev)){
            total_base_r_tkts += prev->process_comp_tkts;
          }

        }  /* end checking for idle_task or RT, and the end of the compensation calculation */
```

Figure 3 Complete block of code showing the compensation ticket calculation.

Now a detailed look at each piece of the block. First, a check is performed to see if the previous process (*prev*) was the idle_task (Figure 4). The previous process is the process that was last running on the CPU. If the previous process was the idle_task, we skip the entire compensation calculation as the idle_task is run only when there are no other runnable tasks, and therefore it is not even considered during a lottery. The lottery scheduler only schedules SCHED_OTHER processes, thus compensation tickets are only granted to SCHED_OTHER processes. The lottery scheduler actually defers to processes that are not SCHED_OTHER (e.g. the real-time class of processes), allowing them to run first. Note that the compensation ticket allocation is also skipped if the SCHED_YIELD bit is set. The reasoning is that if the process is yielding, it does not make sense to increase its likelihood of immediately regaining the CPU by assigning compensation tickets.

```
if(prev == idle_task(this_cpu) || prev->policy != SCHED_OTHER) {
}
```

Figure 4 Checking to see if the previous process was the idle task or if it is a process type other than SCHED_OTHER. In either of these cases the compensation ticket calculation is skipped.

Next some simple tracking of time slice usage is performed (Figure 5). In this implementation, the default time slice was set at 6 ticks of the system clock. This is roughly the same default time slice that the standard Linux scheduler allocates in the kernel version that I operated on. On the PC architecture, 6 ticks are roughly 60 ms, or about 16 possible full time slices per second. Of course the process could be

preempted before using the full time slice, or it could yield the CPU before the time slice expired. This tracking was used to understand how much time slice was being used by processes.

```
/* Track statistics on how much of timeslice was used. */
prev->counter_vals[prev->counter]++;
```

Figure 5 Compensation ticket calculation code section that updates some statistics regarding usage of time slice.

If the full time slice was used, then no compensation ticket allocation is required; however if only a partial was used (Figure 6), then the compensation ticket calculation occurs.

```
/* if entire slice wasn't used */
if(prev->counter > 0) {
```

Figure 6 Testing the value of the *counter* variable to determine if a partial time slice existed when the previous process relinquished the CPU.

The user->user_tpt variable is maintained as processes go on and off the runqueue through the *add_to_runqueue* and *del_from_runqueue* functions. This variable contains the total number of runnable process tickets that a user has. Runnable process tickets are process tickets assigned to currently runnable processes. The lottery scheduler uses this value anytime it needs to convert a process ticket count into a base ticket count. At this point in the *schedule* function the process in question may or may not have left the runqueue, so a conditional is required to

determine if the process is runnable (Figure 7). The value placed in tmp_tpt is necessary for the next task which is to scale process tickets into base tickets.

```
/* If task is still on runqueue then prev->process_tkts are already
   included in user_tpt */
/* The task could still be on the runqueue if its state is still runnable... */
if(task_on_runqueue(prev)){
  tmp_tpt = prev->user->user_tpt;
}
else { /* This is the case where the task got dequeued from the runqueue above */
       /* Since it got dequeued, the user_tpt value was decremented, so we need
          to add the previous processes process tkts back into the user total
          process tickets */
  tmp_tpt = prev->user->user_tpt + prev->process_tkts;
}
```

Figure 7 As part of the compensation ticket calculation, the number of process tickets that a user has allocated to runnable processes is determined.

This conversion from process tickets to base tickets (Figure 8) is necessary as compensation tickets are calculated (and stored) in terms of base ticket value. The conversion partitions the user's base tickets in proportion to the process tickets assigned to runnable processes. So, if a user has only one runnable process, all of the user's base tickets will go to this single process. In extreme cases when a process has a very small proportion of the runnable process tickets, it is possible that the tickets convert to something less than a single base ticket. In this case, the process is simply given a single base ticket. A process can have a very small proportion of the runnable process tickets if the process' owner directly assigns it a small number. It can also end up with a small proportion through a more indirect method; the process' owner may deflate the value of the tickets the process holds by giving other processes many more process tickets.

```
/* tmp_base_cnt is the process's process tix scaled into base tix  */
  tmp_base_cnt = (prev->process_tkts*prev->user->user_base_tickets)/tmp_tpt;
/* be sure that in cases where user_base_tickets is < than tmp_tpt
   we get at least one ticket for the process */
if (tmp_base_cnt < 1) {
    tmp_base_cnt = 1;
}
```

Figure 8 Compensation ticket calculation implementation of a process' process tickets being scaled into base tickets.

Finally, compensation tickets are calculated (Figure 9) in proportion to the amount of time slice that was consumed by *prev*. The calculation is v*t/(t-c)-v where v is the process's ticket value in terms of base tickets, c is the value of the process' *counter* variable, and t is the value that the *counter* variable is set to when a process's time slice is granted. In this implementation t = 6.

```
/* In this case not even a single full tick was used */
if( prev->counter == LOTTERY_SLICE ){
  /* Usage of constant 59 explained in thesis document */
  prev->process_comp_tkts = tmp_base_cnt*59;
}
else {
  prev->process_comp_tkts = ((tmp_base_cnt*LOTTERY_SLICE)/\
  (LOTTERY_SLICE - prev->counter)) -tmp_base_cnt;
}
```

Figure 9 Implementation of compensation ticket calculation.

Since this implementation used the *counter* variable as the measure of time slice usage, there are noteworthy limitations to the degree of precision that is possible in calculating compensation tickets. The *counter* variable is set to the number 6 at the beginning of a process's time slice. This represents 6 ticks of the programmable interval timer. Each tick of the programmable interval timer is 10 milliseconds, and each millisecond of CPU time represents thousands of CPU cycles. Using the

counter variable as a measure of a process's CPU rights does not allow for a high level of granularity. Using something like milliseconds or CPU cycles would allow for a more accurate allocation of compensation tickets. In Waldspurger's Mach kernel implementation (Waldspurger & Weihl, 1994), the time slice was 100 ms, and milliseconds were used to represent CPU share. This allowed Waldspurger to track CPU share down to a level of $1/100^{th}$, where my implementation can only divide the time slice as small as $1/6^{th}$. This makes a difference in compensation ticket calculations. If 100 ms is the time slice, a process's tickets are worth 10 base tickets, and the process uses only 1 of 100 milliseconds, the process will receive (10*100/(100-99))-10 = 990 compensation tickets. (This can be seen at line 1650 of Walspurger's sched_prim.c Mach kernel implementation.) If the same calculation is performed where the *counter* variable's value of 6 is used as the time slice, the largest calculable number of compensation tickets is in the case where *counter* is decremented by 1 and (10*6/(6-5))-10 = 50. The less granular resolution provided by the use of the counter variable results in far fewer compensation tickets being allocated. This will punish worst those processes that use the smallest amount of CPU time. The graph below (Figure 10) shows the number of compensation tickets granted relative to the granularity of the time slice metric.

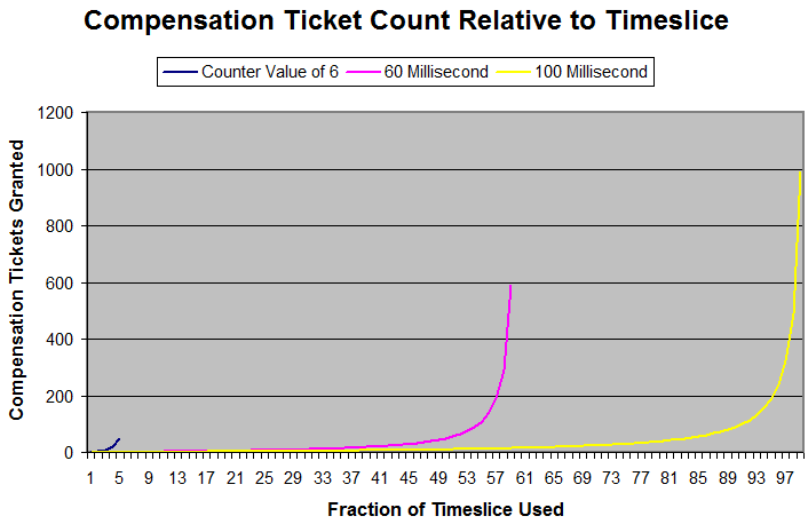

Figure 10 The finer the granularity of the time slice metric, the larger the range of compensation tickets. Example with a process' tickets worth 10 base tickets.

If my implementation used milliseconds as the measure of resource usage, 590 [(10*60/(60-59))-10] compensation tickets would be granted to the process that used a single millisecond or less of its allotted time. As we will see later, it is exactly these short running processes that do perform the worst using this lottery scheduler. Because of this, I decided to give these short-runners the benefit of the doubt, and treat all process that used less than a single tick from the PIT as using one millisecond of the 60 allocated, and v*59 (where v is *base ticket value*) compensation tickets were given to each such process. No further study was performed on the impact of using the *counter* variable as a measure of CPU share.

4.3 Even More Compensation Tickets

As part of this project an additional function, which boosted a short running process's ticket count even beyond the "v*59" modified compensation calculation, was developed (Figure 11).

```
#ifdef CONFIG_PB_LOTTERY
          pwr_raise = prev->counter;
          pwr_boost = 5;
          while (pwr_raise) {
            pwr_boost = pwr_boost*5;
            pwr_raise--;
          }
          prev->process_comp_tkts = prev->process_comp_tkts + pwr_boost;
#endif  /*  CONFIG_PB_LOTTERY */
```

Figure 11 Assigning additional compensation tickets in a manner that is not part of the standard compensation ticket calculation.

This additional function was coined the “I/O enhanced” lottery scheduler as it was designed to help processes that use only a small amount of their time slice. I/O bound processes, such as user interactive tasks, have the characteristic of using only a small bit of their time slice. This I/O enhanced lottery scheduler was compiled into a separate kernel and tested alongside the “standard” lottery scheduler kernel in most experiments. The table below (Table 3) lists the amount of additional compensation tickets that were given to each process based upon prior usage of their time slice.

Table 3 Statically defined compensation ticket allocations that are part of the “I/O enhanced” lottery scheduling kernel.

Consumed Ticks of CPU Time	Bonus Compensation Tickets
6	5
5	25
4	125
3	625
2	3125
1	15625
0	78125

These constant values were selected through a bit of trial and error testing. The only basis for selecting these values was a guess at what would be effective considering the number of tickets and processes that the experiments generally operated with. There was no further research dedicated to tuning these values or the compensation ticket allocation system. Since all the experiments showed a negative bias towards I/O bound processes, this would be a logical area to pursue further.

The *total_base_r_tkts* variable is a global running count of the total number of base tickets for all users' processes that are on the runqueue. Think of the lottery process as the act of selecting a single ticket out of a bin of all tickets that are part of the current drawing. The total number of tickets in the drawing bin will change depending on how many users and processes are competing in each drawing. If a user does not have any processes that need to run (that are on the runqueue) then the user will not participate in the next lottery, and the user's tickets are removed from the drawing bin. The value stored in the *total_base_r_tkts* variable can be thought of as the number of base tickets that will be effective as of the next lottery drawing, or the "*effective base ticket count*". The compensation ticket section of code ends with maintenance of this "*effective base ticket count*" (Figure 12).

```
if(task_on_runqueue(prev)){
  total_base_r_tkts += prev->process_comp_tkts;
}
```

Figure 12 Maintenance of kernel variable that holds sum total number of base tickets for users that have at least one runnable process.

If the process (*prev*), which was just granted some number of compensation tickets, is still on the runqueue, the compensation tickets are added to the "*effective base ticket count*". This count of "*effective base tickets*" consists of both the base tickets held by the users that have at least one process on the runqueue and the compensation tickets held by all processes on the runqueue. Recall that compensation tickets are stored in terms of base ticket value. Thus the "*effective base ticket count*" is total number of base tickets that will be considered in the next instance of a lottery.

Another variable, *user_tpt*, is closely related and is also dynamically maintained. *User_tpt* is a member of the *user_struct* structure. The *user_tpt* variable holds a running count of process tickets for all processes that the user owns that are currently on the runqueue. *User_tpt* is used in the lottery function to scale process's process tickets back into base ticket units.

The maintenance of the *total_base_r_tkts* and the *user_tpt* values is facilitated by the single, simple interface used to maintain the linked list of runnable processes on the system called the *runqueue*. The following function prototypes describe this interface.

```
static inline void add_to_runqueue(struct task_struct * p)
static inline void del_from_runqueue(struct task_struct * p)
```

In addition to these two places where tasks go on and off the runqueue, the variables are maintained in the *schedule* function when compensation tickets are allocated and deallocated, in the *switch_uid* function where process ownership is changed, and in the lottery scheduling system calls where both base ticket and process ticket values are manipulated.

4.4 Holding the Lottery

After the compensation tickets are calculated for the previous process, it is time to select the next process to run. If there are only SCHED_OTHER processes waiting, the lottery is held, a winning ticket is drawn, and the winning process is granted the CPU. Figure 13 shows this implementation of the selection of the next process, the lottery.

```
running_base_cnt = 0;

/* Select ticket nbr. from 1 to total_base_r_tkts */
rand = lottery_random();
lottery_number = (rand%total_base_r_tkts)+1;
lottery_number = lottery_number < 0 ? -lottery_number : lottery_number;

/* Loop through processes on runqueue to do lottery */
/* convert process tickets to base tickets and select winner. */
list_for_each(tmp, &runqueue_head) {
  p = list_entry(tmp, struct task_struct, run_list);

  /* tmp_base_cnt is the process's process tickets scaled into base ticket
     units; plus any compensation tickets that the process holds */
  tmp_base_cnt = ((p->user->user_base_tickets*p->process_tkts)\
                  /p->user->user_tpt) +p->process_comp_tkts;

  /* Need to be sure that in cases where user_base_tickets < user_tpt
     there's at least one ticket for the process */
  tmp_base_cnt = tmp_base_cnt < 1 ? 1 : tmp_base_cnt;

  running_base_cnt += tmp_base_cnt;

  /* unlikely() is the case where we lost tickets via integer division
     truncation, and we want to award the win to the last process in the runqueue */
  if ((running_base_cnt >= lottery_number) || unlikely(tmp->next == &runqueue_head)) {

    if (can_schedule(p, this_cpu)) {
      next = p;
      total_base_r_tkts -= next->process_comp_tkts;
      /* process won... zero compensation tickets */
      next->process_comp_tkts = 0;
      next->lott_wins++;
      break;
    }
    else
    {
      printk("can_schedule failed, not handled in lottery scheduling\n");
      BUG();
    }
  }
}
```

Figure 13 Implementation of the lottery drawing.

Here is an in-depth look at the lottery drawing. Selecting the winning ticket is performed by selecting a random number within the range of 1 up to the total number of "*effective base tickets*" *total_base_r_tkts* (Figure 14).

```
/* Select ticket nbr. from 1 to total_base_r_tkts */
rand = lottery_random();
lottery_number = (rand%total_base_r_tkts)+1;
lottery_number = lottery_number < 0 ? -lottery_number : lottery_number;
```

Figure 14 Selection of a random number that is the winning lottery number.

The Linux kernel has a built-in true random number generator (TRNG) that I was able to use to generate winning lottery tickets. The kernel's random number generator collects random environmental "noise" from device drivers and other sources, and stores this random noise in an "entropy pool". A call to the random number generator retrieves some of these random bits. One problem with this generator was that it was not available early enough in the boot process to service initial calls to the schedule function. Initially I implemented a work around such that the first task on the runqueue was always selected until the random number generator was ready for use. Once it was ready I was able to use it to select random numbers. Another problem with using the built-in random number generator is that the schedule function gets called many times per second, and repeated calls to the TRNG might empty the entropy pool. This could cause a problem for other applications that used a blocking interface to the entropy pool such as /dev/random. This theory was not tested, however I believe it is quite feasible that the first version of the schedule function that used the kernel's TRNG could have emptied the entropy pool. As a

result of these issues I decided to implement a dedicated pseudo-random number generator (PRNG) that would be available in the kernel and to the lottery scheduler. The PRNG I created, in the *lottery_random* function, was based on the example ranqd1 from The Art of Scientific Computing Second Edition (Teukolsky, Vetterling, & Flannery, 1992, p. 284).

The winning process is found by sequentially stepping through the runqueue (Figure 15). There may be many other processes on the system, but only the runnable ones are eligible for the lottery.

```
/* Loop through processes on runqueue to do lottery */
/* convert process tickets to base tickets and select winner. */
list_for_each(tmp, &runqueue_head) {
  p = list_entry(tmp, struct task_struct, run_list);
```

Figure 15 Looping through the runqueue in order to locate the winning process.

As each process in the runqueue is considered, a running total of each process's process tickets scaled into base tickets is maintained in the *running_base_cnt* variable (Figure 16). The kernel does not provide support for floating point math, as a result, the integer division in this block of code proved to be problematic. Any fractional remainder of the division was lost due to truncation. The consequence of this was that lottery tickets were lost and the running total of effective base tickets did not add up to the total number of effective base tickets. When the winning lottery number was near or at the top of the lottery ticket range, i.e. nearly equal to the total number of effective base tickets; the lottery drawing could result in no winner if too many tickets were lost due to truncation.

```
tmp_base_cnt = ((p->user->user_base_tickets*p->process_tkts)\
                /p->user->user_tpt) +p->process_comp_tkts;

/* Need to be sure that in cases where user_base_tickets < user_tpt
   there's at least one ticket for the process */
tmp_base_cnt = tmp_base_cnt < 1 ? 1 : tmp_base_cnt;

running_base_cnt += tmp_base_cnt;
```

Figure 16 Process tickets are scaled into base tickets because the lottery is held in terms of base tickets.

When the running total is greater than or equal to the selected winning ticket, the winning process has been found. An "unlikely" case is introduced (Figure 17) to handle the situation discussed previously where lost lottery tickets result in a lottery being held with no winner selected. Note, if no winner was selected, the idle_task would be selected by default. This would result in a runnable task sitting on the runqueue, while the idle_task was incorrectly selected. To ensure this does not happen, the second part of the "if conditional" checks to see if the end of the runqueue has been reached. If the end of the runqueue has been reached without selecting a winner, then the last task in the runqueue is awarded the lottery win.

```
/* unlikely() is the case where we lost tickets via integer division
   truncation, and we want to award the win to the last process in the runqueue */
if ((running_base_cnt >= lottery_number) || unlikely(tmp->next == &runqueue_head)) {
```

Figure 17 Conditional that identifies the process that wins the lottery.

Figure 18 consists of some lottery scheduling book-keeping that is performed before the next process begins to run. The winning process *p* is chosen as the process *next* that is to be run. The count of effective base tickets is updated before the process's compensation tickets are zeroed. The compensation tickets are zeroed because the

process is only granted compensation tickets until its next lottery win. Once a winner is found, the loop through the runqueue is terminated, and further preparations are made to switch to the *next* task.

```
next = p;
total_base_r_tkts -= next->process_comp_tkts;
/* process won... zero compensation tickets */
next->process_comp_tkts = 0;
next->lott_wins++;
break;
```

Figure 18 Final lottery scheduling book-keeping performed before the “next” process starts running.

5. Experiments

The following experiments focus on testing both the purported strengths of the lottery scheduler as well as the overall performance of the scheduler as compared to the standard Linux scheduler. The experiments range from standard benchmarks to a practical application test controlling web server performance with the scheduler. An area of particular focus is the scheduler's capacity to manage CPU bound and I/O bound processes.

5.1 Description of Development and Testing Environment

Suse Linux 9.0 was the Linux distribution that was used for the project. The distribution's default kernel version 2.4.21-99-default was replaced with a vanilla 2.4.31 kernel. The 2.4.31 kernel was also the version that the lottery scheduler was implemented in. A vanilla 2.4.19 kernel with Rik van Riel's "FairSched" patch was selected as a third kernel to compare. In order to keep as many factors similar, the same .config file kernel compile options were used for the standard lottery, the I/O enhanced lottery, and the vanilla 2.4.31 kernels. Uniprocessor only support, for example, is selected in all cases. The lottery scheduling implementation does not support SMP. To test and generate graphs of test results tools like httperf, gnuplot, ps2pdf, expect, autobench, and shell script were used. The hardware used to test the modified kernels was an Intel Pentium 4 (2.4 GHz), with 512 MB of RAM. In the case of an Apache Web Server test, three other PCs were used to issue the http requests to the PC running the apache web servers. The three http clients were

Pentium III class PCs with 256 to 384 MB of RAM. Networking between the systems consisted of wired 100M-bit Ethernet cards run through a NetGear FS108 Switch.

5.2 Controlling Relative Rate of Process Execution

This test compares the abilities of the Linux kernel's default scheduler, and the lottery scheduler to control the relative rate of execution (RRE) of processes owned by a single user. The lottery scheduler has controls designed for controlling relative rates of execution. Specific rates are supposed to be achievable by simply assigning corresponding ratios of process tickets to processes. The standard Linux kernel provides the *nice* process attribute that can be used to control execution rates. The expectation is that it will be easiest to control the relative rate of execution using the lottery scheduler. A description of the test framework, results and analysis follows.

The test runs two CPU bound processes at the same time, measuring and graphing their progress. From the graphed results conclusions can be drawn regarding a scheduler's efficacy in maintaining control over RRE. The test program is comprised of a shell script harness and several CPU bound test programs. The harness takes multiple parameters that establish the conditions of the test. The CPU bound programs are launched at as close to the same time as possible in order to make the graphs of their progress comparable. The programs are launched under a user account that does not have any other processes running on the system. This is done so the execution rates of the test processes are not competing with other processes that are not part of the test.

The CPU bound programs burn cycles with a series of loops. The inner most loop performs some math calculations, while the outer most loop keeps track of elapsed time (via *gettimeofday()*) between each iteration of itself and prints (via printf()) these times to a text file. The values in the text file are then used to graph the processes' progress relative to the passage of time.

Parameters given to the test harness specify different values for the execution counts of these inner and outer loops. For example, a test can be setup to run with a single iteration of the outer loop and 100 iterations of the inner loop. This test would perform the same amount of math calculations as a test setup to run 100 iterations of the outer loop and a single iteration of the inner loop. Parameters can also specify that additional CPU and/or disk I/O load should be applied to the system as well. The intention is to see how the additional load impacts the scheduler's ability to control execution rates.

Below is a graph (Figure 19) that displays the results of a run of the test. The parameters of the test run are displayed in the title block of the graph. In this example, the standard lottery scheduling kernel is being tested, there has been no additional system load applied, and the ticket ratio between the two processes is 4:1. The results are the average of 6 separate runs. The inner and outer loop counts are a measure of work completed during the test. The 4:1 ratio selected for this test means that one process was expected to execute four times faster than the other and the graph shows that the results match the expectations. Process #1 with 200 tickets has only completed 250 units of work by the time Process #2 completed 1000 work units.

Once Process #2 finishes 1000 units of work and exits, Process#1 is given all of the CPU and finishes in short order.

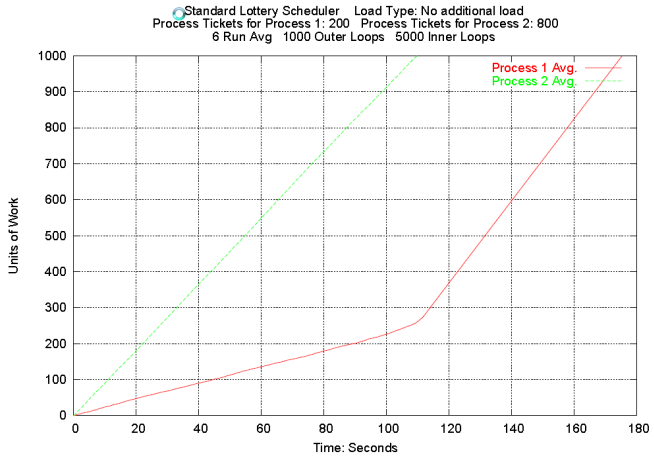

Figure 19 Graphical representation of two CPU bound processes with a 4 to 1 lottery ticket allocation. Process 1 is only ¼ complete with its task when Process 2 finishes.

Figure 20 displays a similar test but at a 3:1 ratio. Once again the lottery scheduler appears to do an excellent job of controlling relative rate of execution.

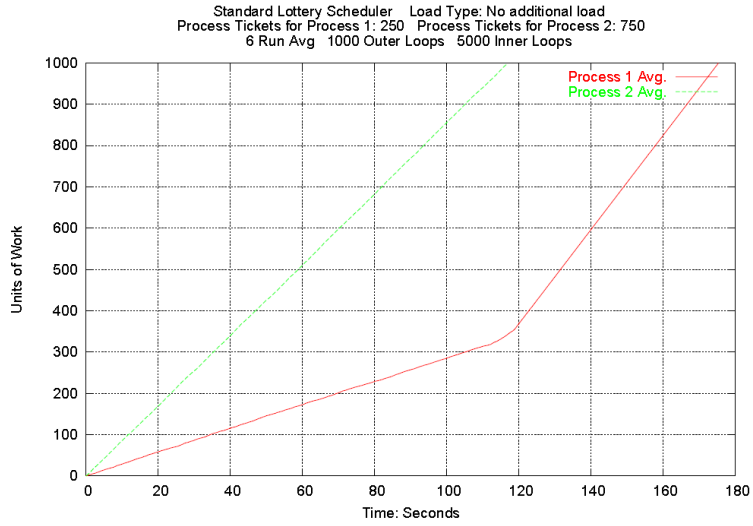

Figure 20 Two CPU bound processes with a 3 to 1 lottery ticket allocation. Process 1 is only 1/3 complete with its task when Process 2 finishes.

The Linux *nice* process attribute can also be used to control rate of execution. It took a good bit of experimentation to determine the nice values required to attain a specific relative rate of execution, and in all cases it is not as easy to use as the lottery scheduling mechanism. This test targets a 4:1 ratio to be attained through the *nice* mechanism. The results are consistent, but are not exactly 4:1. After a good bit of experimentation, the nice values of 0 and 16 were found to produce a ratio on the high side of 4:1, while the nice values of 0 and 17 were found to produce a ratio just below 4:1. Below are two graphs, one shows test results with *nice* values of 0 and 16 (Figure 21) and the other shows results with values of 0 and 17 (Figure 22).

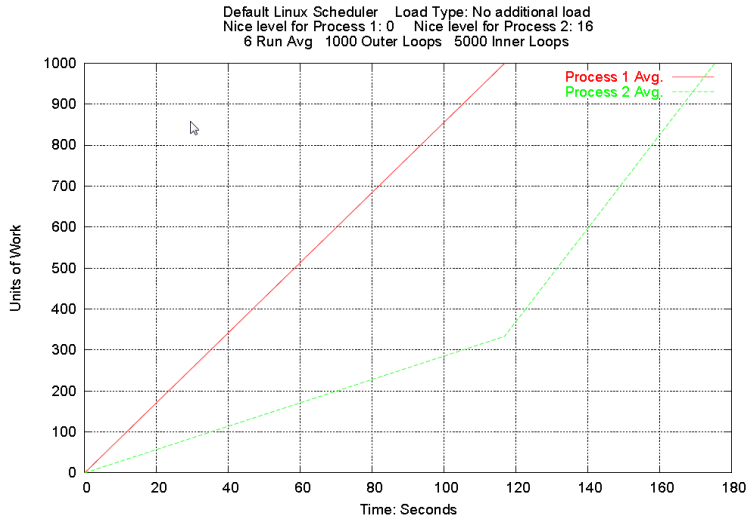

Figure 21 Two CPU bound processes with *nice* values of 0 and 16. Process 1 has the *nice* value of 0 and Process 2 the *nice* value of 16.

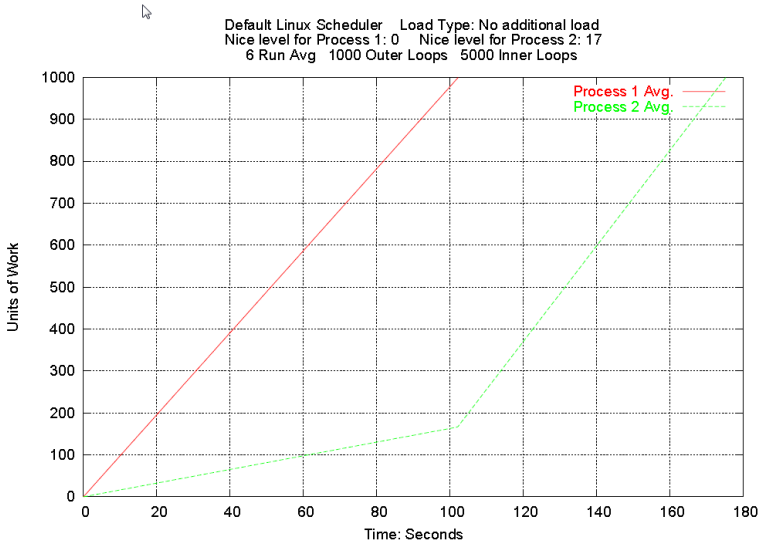

Figure 22 Two CPU bound processes with *nice* values of 0 and 17. Process 1 has the *nice* value of 0 and Process 2 the *nice* value of 17.

While rates of execution can be controlled with the default Linux scheduler and the *nice* property, it is not easy to determine the *nice* values necessary to attain specific

ratios. It is even more difficult as the number of processes in the test set increase. Compare this to the lottery scheduler with which ratios are naturally specified through ticket allocations.

The lottery scheduler allows for dynamic reallocation of lottery tickets to achieve respecification of execution rates after processes have already started. While it is also possible to alter execution rates of running process with the Linux *renice* function, determining values that result in specific rates is again difficult. Below is a graph that shows the lottery scheduler changing execution rate of a running process which alters the relative rate of execution between the two processes from 4:1 to 1:1 (Figure 23). Process #2 runs at a rate 4 times that of Process #1 until halfway through its 1000 work units. At that point the process tickets of Process #2 are changed from 800 to 200. This process ticket change makes both processes progress at the same rate until Process #2 finishes.

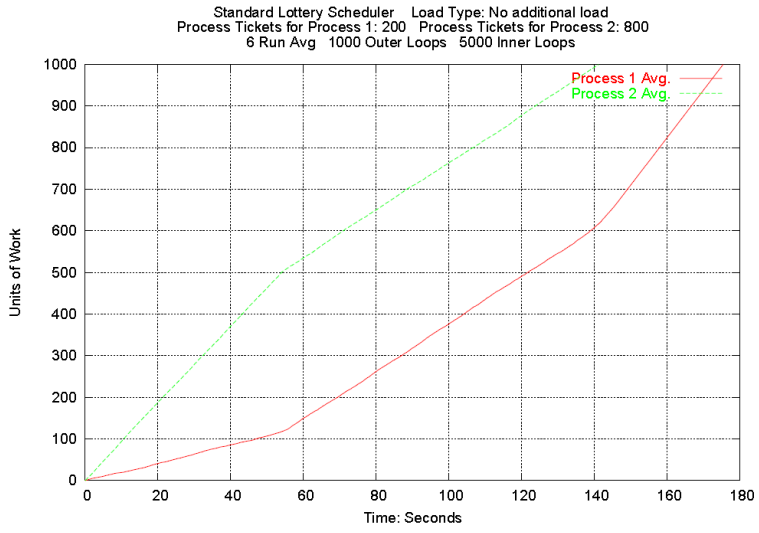

Figure 23 Lottery ticket ratios being changed mid-test from 4:1 to 1:1.

The results displayed so far are all from tests that ran well over 100 seconds and that were averaged with the results from multiple test runs. If similar tests are run over shorter periods of time and as single instances rather than multiples averaged, a clear difference in the progression of execution can be seen. The lines of the graph that describe process execution have a definite wobble in the case of the lottery scheduler, suggesting uneven execution. In the case of the default Linux scheduler the lines are even and steady suggesting that processes are making more regular progress. Below in the graph of a single run rather than 6 runs averaged together, this "wobble" can just barely be seen (Figure 24).

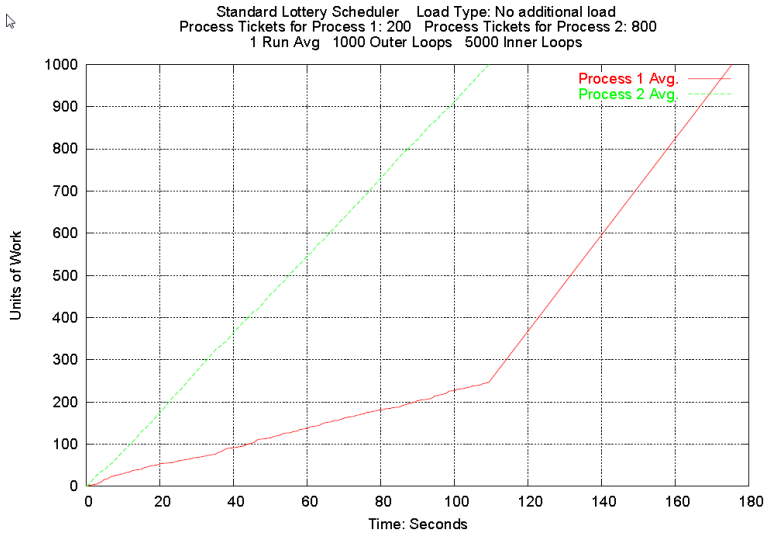

Figure 24 Display of a single run rather than 6 averaged together. The "wobble" in the graph lines denoting execution progress starts to become visible.

Zooming in further by decreasing the amount of work that is to be performed, this characteristic is even more evident (Figure 25).

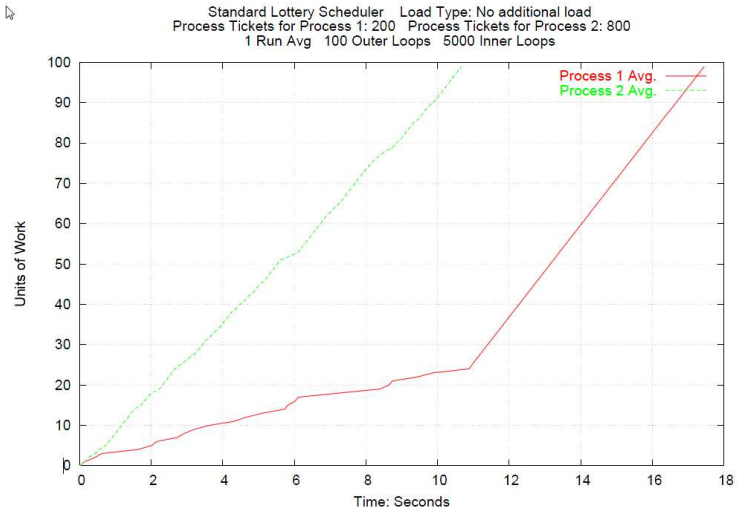

Figure 25 "Wobble" is even more visible in this graph of a single run with a reduced workload of 100 outer loops.

Contrast that with the graph below from the default Linux scheduler which features even lines representing steady execution (Figure 26). Now a closer look into this anomaly.

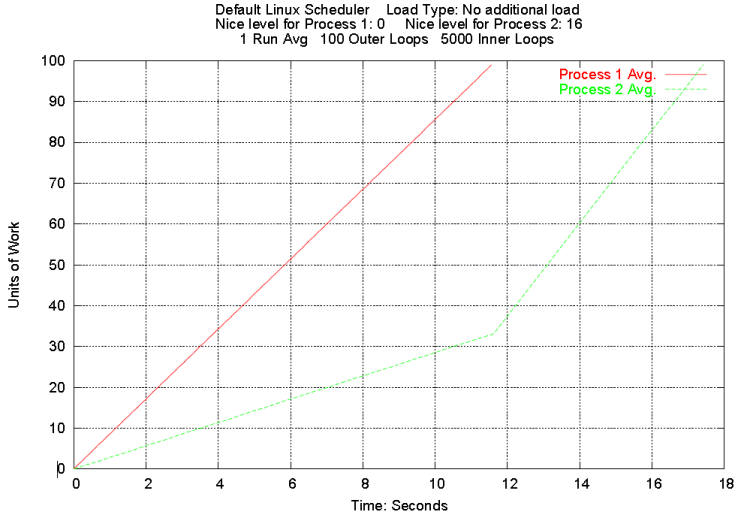

Figure 26 The standard Linux scheduler shows no wobble in this graph of a single run with a reduced workload of 100 outer loops.

To “zoom up” even further, the length of the inner loop can be reduced (from 5000 to 500), resulting in a reduction of the overall length of the test, and in more frequent reporting from the outer loop. As seen in the graph below, the lottery scheduler shows inconsistent relative rates of execution over periods of time less than a few seconds (Figure 27). This appeared to be the result of probability over the short-run, the rational being that in the short term a process can win the lottery in an amount that is disproportionate to the process’s ratio of lottery tickets. However, over the long term probability will stabilize and ticket ratios will be properly represented in execution rates.

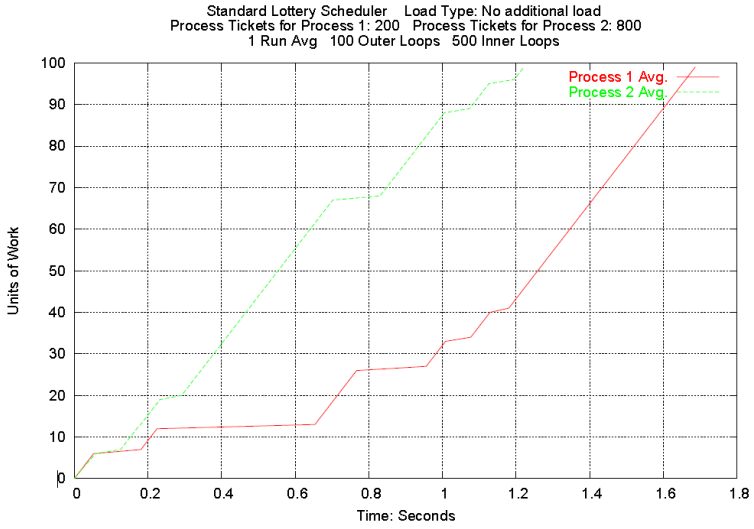

Figure 27 Further reducing the workload to 500 inner loops with the lottery scheduler makes the execution progress anomaly even more visible.

To investigate further, profiling code was added to the lottery scheduling kernel. The profiler ran at each instance of the lottery, and recorded the following metrics.

- **Winning Process** - The process that won the lottery.
- **TBLT** - The total base lottery tickets (including compensation tickets) of all runnable processes on the system, at the time of the lottery
- **WLTN** - The winning lottery ticket number
- **WPCT** - The number of compensation tickets held by the winning process
- **WPTEBT** - The winning process's total effective base tickets, which is the process's process tickets scaled into base tickets, plus any held compensation tickets.

Table 4 contains the profiling data for the two processes from the test run displayed in Figure 27. This is the data for *just* the two processes involved in the test although there were other processes running on the system at the same time.

Table 4 Profiling data from the lottery scheduler for the test run displayed in Figure 27.

Row #	Winning Process	TBLT	WLTN	WPCT	WPTEBT
1	Process 2	1000	538	0	800
2	Process 1	1000	961	0	200
3	Process 2	1040	189	0	800
4	Process 2	1040	634	0	800
5	Process 2	1040	395	0	800
6	Process 2	1040	676	0	800
7	Process 2	1040	169	0	800
8	Process 2	1040	334	0	800
9	Process 2	1040	727	0	800
10	Process 1	1040	952	40	240
11	Process 1	1000	875	0	200
12	Process 2	1000	722	0	800
13	Process 2	1000	779	0	800
14	Process 2	1000	772	0	800
15	Process 1	1000	881	0	200
16	Process 2	1000	502	0	800
17	Process 1	1000	856	0	200
18	Process 2	1000	269	0	800
19	Process 2	2600	1309	1600	2400
20	Process 2	2600	1623	1600	2400
21	Process 2	108200	41558	47200	48000
22	Process 1	1000	331	0	1000
23	Process 1	1000	323	0	1000
24	Process 1	1000	340	0	1000
25	Process 1	1000	161	0	1000
26	Process 1	1000	562	0	1000
27	Process 1	1000	937	0	1000
28	Process 1	1000	624	0	1000
29	Process 1	1000	85	0	1000
30	Process 1	1000	178	0	1000
31	Process 1	32000	437	500	1500

To help visualize the ordered list of lottery wins for each process, a bar-chart representing the order of wins is placed under the chart that tracks the rate of execution in Figure 28. Note that the alignment of the ordered lottery win data on the bar chart and the sections of the line chart that slope upward (representing process execution) is almost perfect. It is evident that the random nature of the lottery results in unpredictable execution patterns in the short-term.

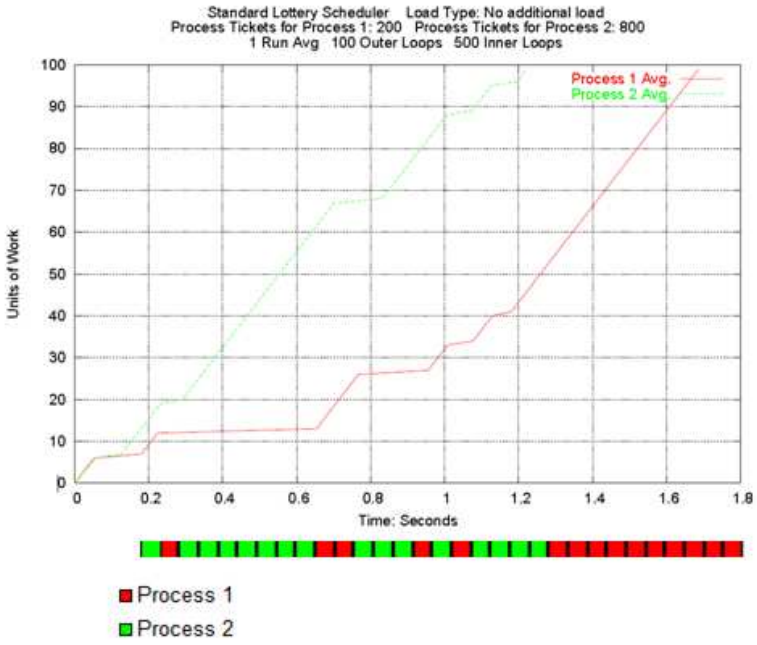

Figure 28 Bar chart displaying ordered list of lottery wins placed under chart tracking the rate of execution.

The profiling data provides some additional insights to how the lottery scheduler operates. Table 5, below, contains a subset of the data displayed in Table 4.

Table 5 Profiling data provides insight to the operation of the lottery scheduler.

Row #	Winning Process	TBLT	WLTN	WPCT	WPTEBT
1	Process 2	1000	538	0	800
2	Process 1	1000	961	0	200
3	Process 2	1040	189	0	800
4	Process 2	1040	634	0	800
5	Process 2	1040	395	0	800
6	Process 2	1040	676	0	800
7	Process 2	1040	169	0	800
8	Process 2	1040	334	0	800
9	Process 2	1040	727	0	800
10	Process 1	1040	952	40	240
11	Process 1	1000	875	0	200
12	Process 2	1000	722	0	800
13	Process 2	1000	779	0	800

At Row #2 the Total Base Lottery Ticket count was 1000. This means that likely only Process 1 (200 tickets) and Process 2 (800 tickets) were available to run and they were the only processes competing in the lottery. Since Process 1 won with the lottery number of 961, it is apparent that Process 2 was closest to the head of the run queue. This is the case because the lottery algorithm searches for the winner from the start of the run queue. So if Process 2 has 800 tickets and is at the head of the run queue, Process 1 needs a lottery ticket with a number over 800 to win. At Row #3, Process 2 won, and there were 1040 Total Base Lottery Tickets. The additional 40 tickets were likely compensation tickets given to Process 1. This means that Process 1 did not use its entire time slice (6 ticks of the clock) during its turn after winning at Row #2. Further evidence that the 40 extra base tickets are compensation tickets belonging to Process 1, exists in Row #10 where Process 1 won with a ticket number of 952 and was holding 40 compensation tickets when the win happened. Note that after the win at Row #10, the Total Base Lottery Ticket count goes back down to

1000, and when Process 1 won again in Row #11, it was not holding any compensation tickets.

Table 6 below, is another subset of the data displayed in Table 4. Process 2 has 1600 compensation tickets in both rows #19 and #20. This is likely the case where only two of the process' six clock tickets were used. In row #21 Process 2 has 47200 compensation tickets. This is the case where not even a single tick of the clock was used in the process's prior turn. As explained in the design of the lottery algorithm used for this project, this is a special case where an extra large allocation of compensation tickets is assigned. Also on Row #21 TBLT, the total number of base tickets held by runnable processes at the time of the lottery, is 108200. This is most likely because another process (not one of the test processes) was also holding a very large number of compensation tickets at the time.

Table 6 Lottery scheduling profiling data that includes some large compensation ticket allocations.

Row #	Winning Process	TBLT	WLTN	WPCT	WPTEBT
17	Process 1	1000	856	0	200
18	Process 2	1000	269	0	800
19	Process 2	2600	1309	1600	2400
20	Process 2	2600	1623	1600	2400
21	Process 2	108200	41558	47200	48000
22	Process 1	1000	331	0	1000
23	Process 1	1000	323	0	1000

5.2.1 RRE with Additional Load

Similar RRE (relative rate of execution) tests were run in the presence of varying system loads. Below are the results of tests run with additional CPU load.

The CPU load was generated by a process that performed math functions in an endless loop. This process running on the system alone takes the CPU to %100 busy. The results of the standard Linux kernel's scheduler look very similar to the results with no load. In the graph displayed below the approximate 5:1 execution rate ratio is apparent with nice values of 0 and 17 being used again (Figure 29). The only difference between this test run and a run with no load is the overall time it takes to finish. Note, the additional load was run at nice level 0 so Process 2 runs the entire test at a significant disadvantage.

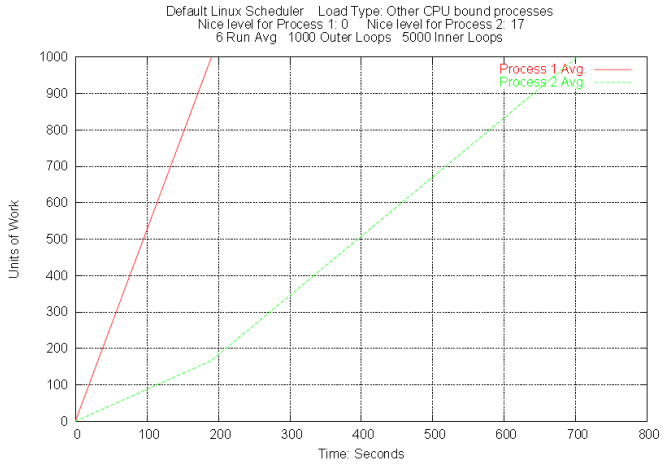

Figure 29 RRE test with additional CPU load. This is tested with the standard Linux scheduler, and *nice* value assignments of 0 and 17.

Under additional CPU load, the results for both the standard lottery scheduler and the lottery scheduler enhanced for I/O were the same as the results from the test with no load. The results for the two lottery schedulers are graphed below in Figure 30 and Figure 31. For the test results displayed here, the additional load was run under fred's user account while the test processes were run under bill's account. However, the

results were the same even if the additional load was run under bill's user account along with the test processes. As long as Process 1 and Process 2 had 4:1 ticket allocations the resulting RRE was 4:1.

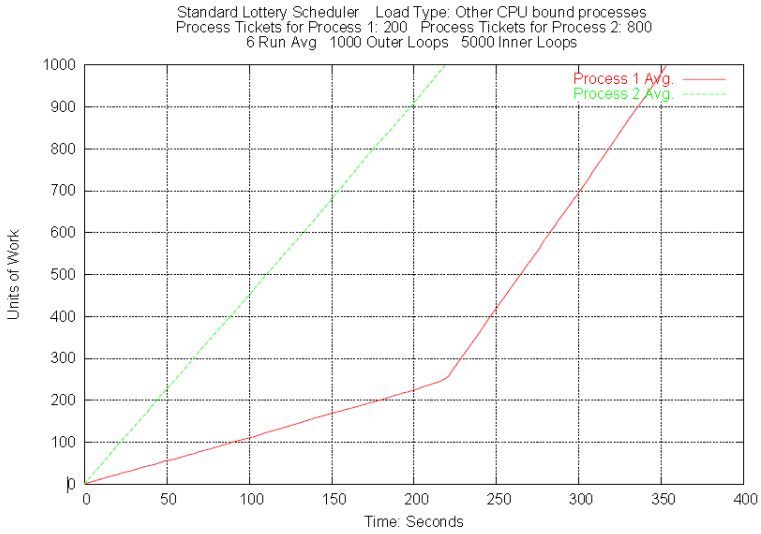

Figure 30 RRE test with additional CPU load. This is tested with the standard Lottery scheduler, and lottery ticket allocation in a 4:1 ratio.

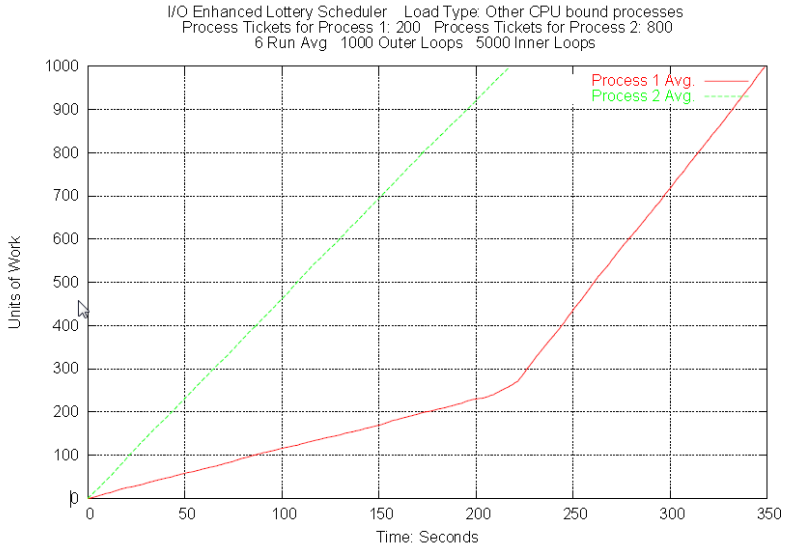

Figure 31 RRE test with additional CPU load. This is tested with the I/O Enhanced Lottery scheduler, and lottery ticket allocation in a 4:1 ratio.

The same tests were then run with additional I/O load placed on the system. The I/O load that was applied was a continuous loop of mixed disk reads and writes. The files read from and written to were larger than physical RAM to prevent the entire file from sitting in cache. The results with the I/O load were the same as the results with the additional CPU load. The additional load did not impact any of the three schedulers' ability to hold relative rate of execution.

In summary, the results of these tests show that the relative rate of execution between processes is easiest to specify with a lottery scheduler. While a highly precise RRE can be held by the default Linux kernel's scheduler, it is not possible to specify an RRE with a high degree of accuracy. The lottery scheduler can be directed to hold an RRE with a higher degree of accuracy, but a lower degree of precision.

5.3 Test of "User Insulation" capabilities

As previously discussed, another key feature of lottery scheduling is the scheduler's capability to preserve resource rights across trust boundaries. In my implementation a prime example of a trust boundary is the user account. This experiment tested the scheduler's ability to preserve resource rights between users. The CPU resource rights were doled out in the form of base tickets. This experiment used a setup very similar to the one used in the RRE tests. The test processes were CPU bound jobs and the test results were again displayed as graphs that chart the processes' progress relative to each other. The differences between this experiment and the RRE test were that a second user account was being used, and base ticket count rather than process ticket count was being manipulated. In describing these test results "base tickets" may be referred to as "user tickets" in some contexts.

In the first "user insulation test", each user was granted the same number of base tickets. User bill started a single CPU bound process while user paul started two CPU bound processes. All processes were started with a similar number of process tickets. Since paul's two processes were started with the same number of process tickets, they ran at the same rate relative to each other. Since bill had only one process the process ticket count did not even matter. Any one of the processes alone would push the CPU to 100% usage. Proper preservation of resource rights would result in bill's process finishing when paul's 2 processes were only 50% finished. Below are the results of where bill had a single process running and paul had two (Figure 32). Note that bill's process completed the full 1000 units of work while paul's two processes had only completed 500 units of work.

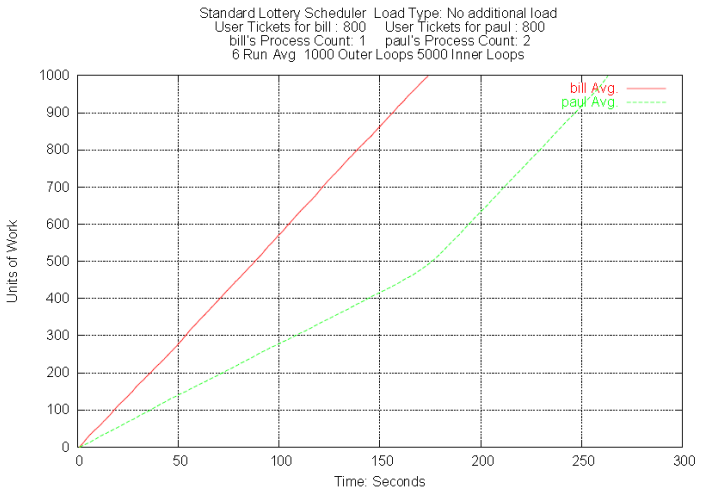

Figure 32 Graph displaying the results of a user resource rights insulation test in which user paul had twice as many processes as user bill.

In a more extreme case this was tested with user paul starting 10 processes while user bill started only one again. As seen in the graph below, bill's single process finished when paul's 10 processes were only 10% complete (Figure 33). The lottery scheduler appears to have very strong fair-share characteristics when scheduling CPU bound processes.

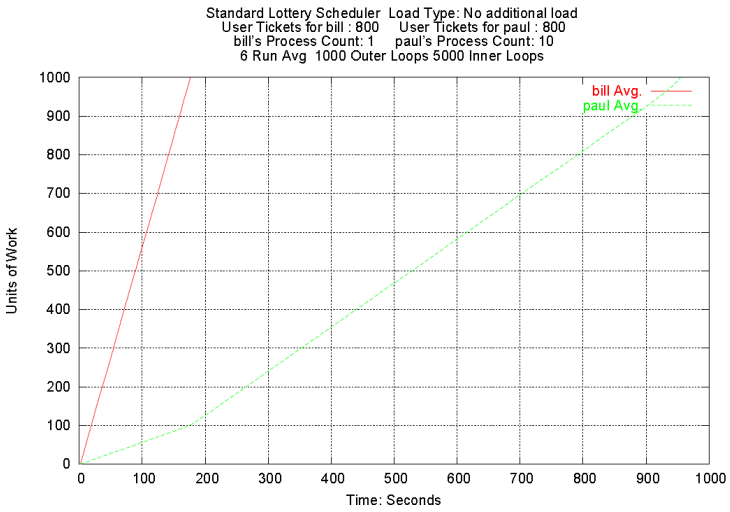

Figure 33 Graph displaying the results of a user resource rights insulation test in which user paul has 10 times as many processes as user bill.

When performing the same test with the default Linux scheduler, the expected result was that all processes would run at roughly the same rate. The goal of the default Linux scheduler is to maximize the throughput of all processes equally. As seen in the graph below, this is exactly what happened (Figure 34). All processes' progress tracked equally, meaning that there was not any notion of fair-share in the default Linux scheduler. The default Linux scheduler balances the CPU well across all processes, but not across all users.

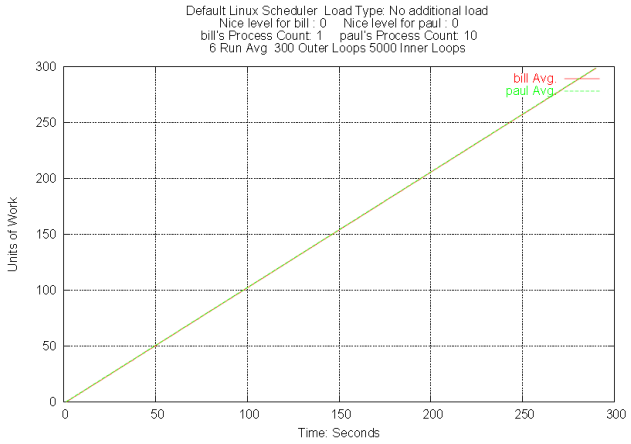

Figure 34 Graph displaying the results of a user resource rights insulation test with the standard Linux scheduler in which user paul has 10 times as many processes as user bill.

Another fair-share scheduler was also tested in this user resource rights insulation experiment. The kernel tested was a vanilla 2.4.19 kernel patched with Rik van Riel's FairSched patch. Below is a test run at a 2:1 process count ratio, and the fair-share scheduler does a good job of balancing user access to the CPU resource (Figure 35).

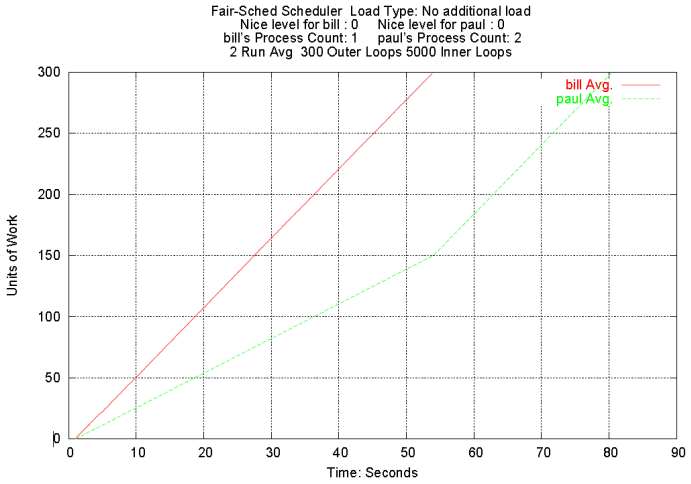

Figure 35 Graph displaying the results of a user resource rights insulation test with Rik van Riel's FairSched scheduler in which user paul has twice as many processes as user bill.

Again at a 10:1 process count ratio, the FairSched scheduler also appears to keep the appropriate balance as bill's single process goes through 300 units of work in the same time that paul's 10 processes go through about 30 units of work each.

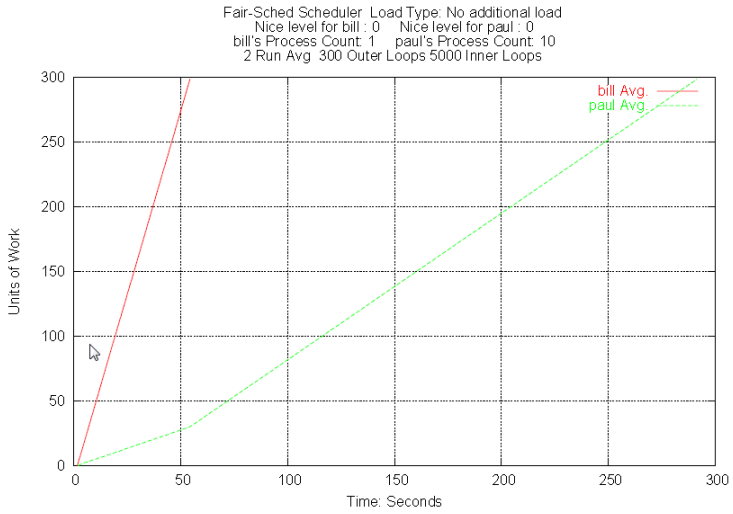

Figure 36 Graph displaying the results of a user resource rights insulation test with Rik van Riel's FairSched scheduler in which user paul has 10 times as many processes as user bill.

As seen in Figure 37 and Figure 38 below, even in an extreme case of 60:1 process count ratio both the FairSched scheduler and the standard lottery scheduler perform as expected.

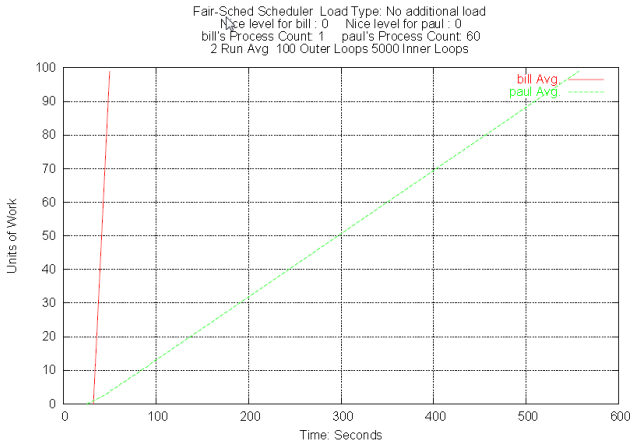

Figure 37 Graph displaying the results of a user resource rights insulation test with Rik van Riel's FairSched scheduler in which user paul has 60 times as many processes as user bill.

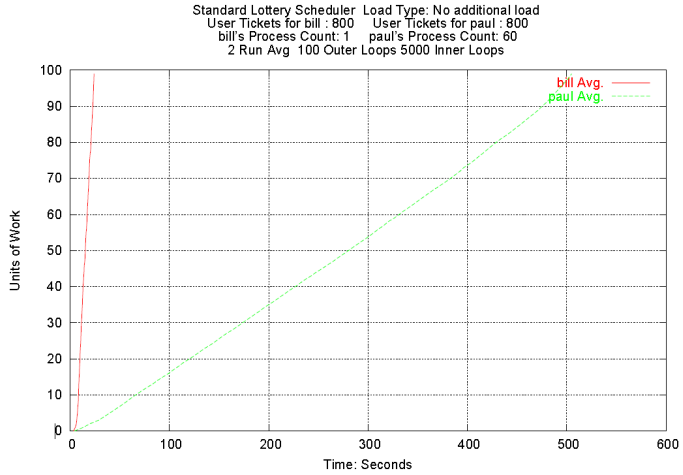

Figure 38 Results of a user resource rights insulation test with the standard Lottery scheduler in which user paul has 60 times as many processes as user bill.

5.4 Scheduler Comparison using Interbench

Interbench is a benchmark utility that attempts to simulate the action of several different interactive processes, such as the X window system, a process playing audio, a process performing video playback, and a gaming process. These interactive process are run with no load and then with a variety of different loads, each for 120 second intervals. The simulated processes and additional load processes span the spectrum from fully CPU bound to highly interactive. Interbench 0.31 was used, and can be found at http://users.on.net/~ckolivas/interbench/.

Processes deemed interactive have the characteristic of using just a portion of their allotted time slice, yet requiring access to the CPU on a frequent basis. Lottery scheduling is designed to provide interactive tasks with sufficient access to the CPU through the compensation ticket mechanism. When a process uses only a small

proportion of its time slice, it is temporarily given extra lottery tickets (compensation tickets) in an amount inversely proportional to the amount of time slice used. This means that if a process uses only a small piece of its time slice, it is given more of a chance to win a future lottery. These extra lottery tickets are used in subsequent lotteries until the process wins again. Once the process wins another lottery, the compensation tickets are relinquished.

The Interbench benchmark was tested on the standard Linux 2.4.31 kernel, the standard lottery scheduling kernel and the I/O enhanced lottery scheduling kernel. The FairSched kernel was not used in this test because the test is run in single user mode. With only a single user there is no CPU hogging user to protect other users from. The Interbench test was run in single user mode to minimize the impact of other processes. When running the test with a lottery scheduling kernel, all processes were assigned the same number of lottery tickets. This means that when comparing the performance of the lottery kernel against the standard Linux kernel per-process ticket allocation is not a factor. Rather the results should measure the efficacy of the compensation ticket mechanism and compare overhead between the two kernels.

The purpose of the Interbench benchmark tool is to compare how different kernel configurations perform for interactive tasks. In the tests performed for this project, the benchmark's measurement of scheduling latency was tracked. Scheduling latency is defined as the amount of time that passes between a process becoming ready to be scheduled and actually being scheduled. The lottery win profiling performed as part of the RRE test showed that scheduling latency can be extreme at times, so the expectation is that the lottery scheduling kernels will display

larger maximum latency measurements than the standard Linux scheduler, but the average latency should be in-line with the standard Linux scheduler. Below the results of each benchmark test are analyzed.

5.4.1 X Windows Simulation

The Interbench X windows simulation uses a variable amount of CPU ranging from 0 to 100 percent. This is supposed to imitate a window being dragged across the screen and dropped. Below is a chart (Figure 39) that compares the average scheduler latency from the three kernels. Each category along the Z-axis represents a kernel and each series along the X-axis represents some type of additional load. The average latency is measured in milliseconds, and lower latency is desirable in terms of smooth performance from a user's perspective (interactivity). The standard Linux scheduler did a superior job of keeping low latencies when there was other CPU bound load present. The CPU Burn load is the most CPU bound process. The Compile Load is also fairly heavily CPU bound. When there was an interactive or I/O bound load present the lottery schedulers' latencies tended to be lower. The Video Playback load is an example of an interactive load.

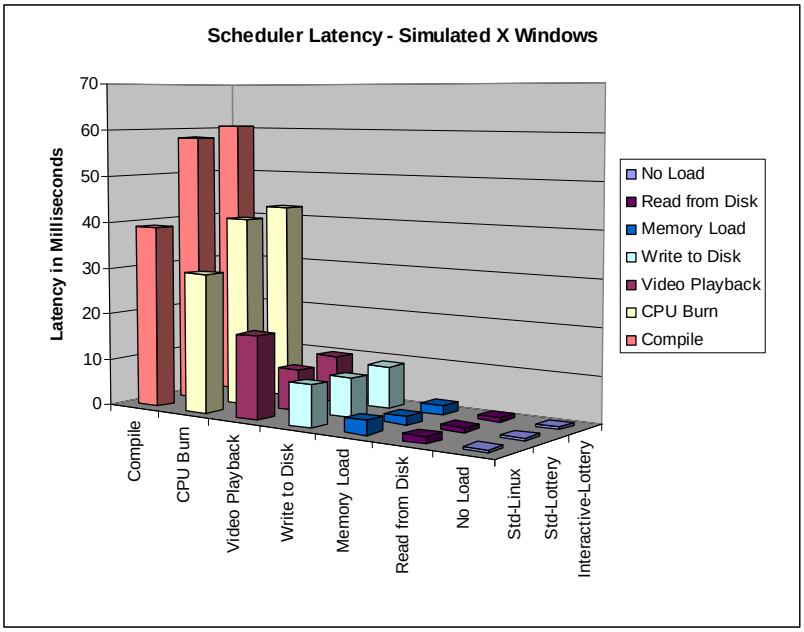

Figure 39 For the simulated X Windows benchmark, latency is lower with the Standard Linux scheduler in the presence of the CPU intensive additional loads Compile and CPU Burn. Latency is lower with either Lottery scheduler in the presence of I/O intensive additional loads, Video Playback, Memory Load, and Read from Disk.

5.4.2 Video Playback Simulation

The Interbench simulation of video playback uses a process that tries to receive CPU 60 times per second and uses 40% CPU. In the results below (Figure 40) video playback is benchmarked with a variety of different loads. The video playback performs better with lottery scheduling only in the presence of even more heavily interactive loads. "Memory Load", which simulates heavy memory/swap pressure, and "Read from Disk" are the only cases where the lottery scheduler's latency is lower. Whenever there is a more CPU bound load running at the same time the latency times for the video playback process being benchmarked go up with either

lottery scheduler. Both of the lottery schedulers heavily favor CPU bound loads. Note that the IO-enhanced lottery scheduler (labeled Interactive-Lottery) services the simulated video playback process better than the standard lottery scheduler when running along with CPU bound load.

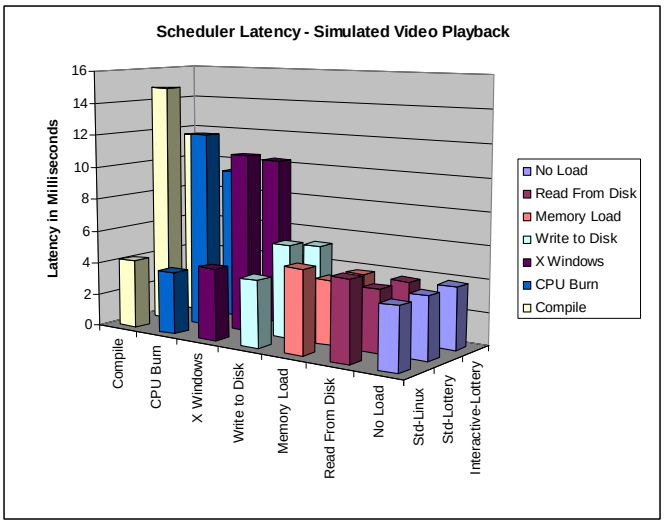

Figure 40 Latencies for the relatively interactive Simulated Video Playback process are low for the standard Linux scheduler when running with CPU bound load. Latencies are only favorable for the lottery schedulers when running together with the most I/O intensive loads.

5.4.3 Gaming Simulation

Even though "gaming" is an interactive application from the end user perspective, the Interbench gaming simulation process is not executed with traditional interactive characteristics. The gaming simulation simply tries to get as much CPU as it can. This is supposed to replicate the behavior of a CPU bound game. So far it has been clear that the lottery schedulers favor CPU bound processes; so it stands to reason that latencies for this CPU hungry benchmark process should be lower with

the lottery schedulers. The results are generally consistent with this expectation (Figure 41).

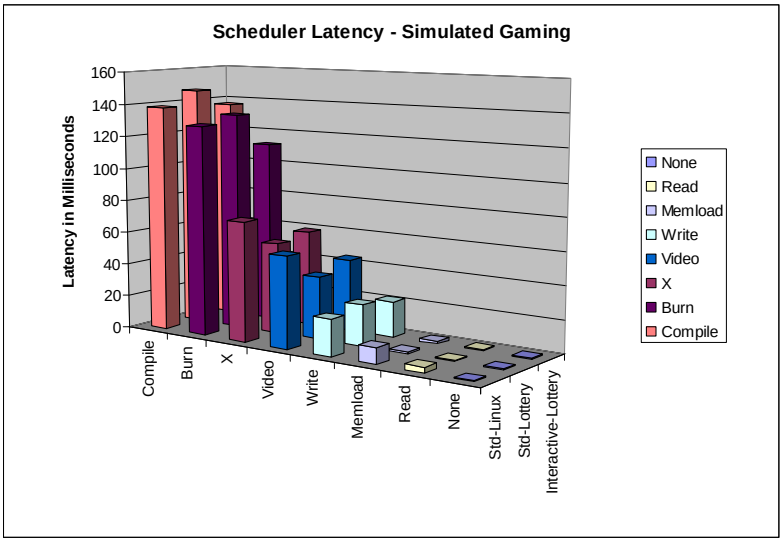

Figure 41 Simulated Gaming is a CPU hungry benchmark. Lottery scheduler latencies measured are generally equal to or less than those of the standard Linux scheduler; consistent with the apparent lottery schedulers' bias toward CPU bound processes.

5.4.4 Audio Playback Simulation

Audio is the most interactive of all the simulated processes. It tries to run every 50ms and uses 5% CPU when it runs. As expected both lottery schedulers perform much worse that the standard Linux kernel's scheduler (Figure 42). As in all test cases where the benchmark process is highly interactive, the lottery scheduler enhanced for interactive processes does provide marginally better latencies than the standard lottery scheduler.

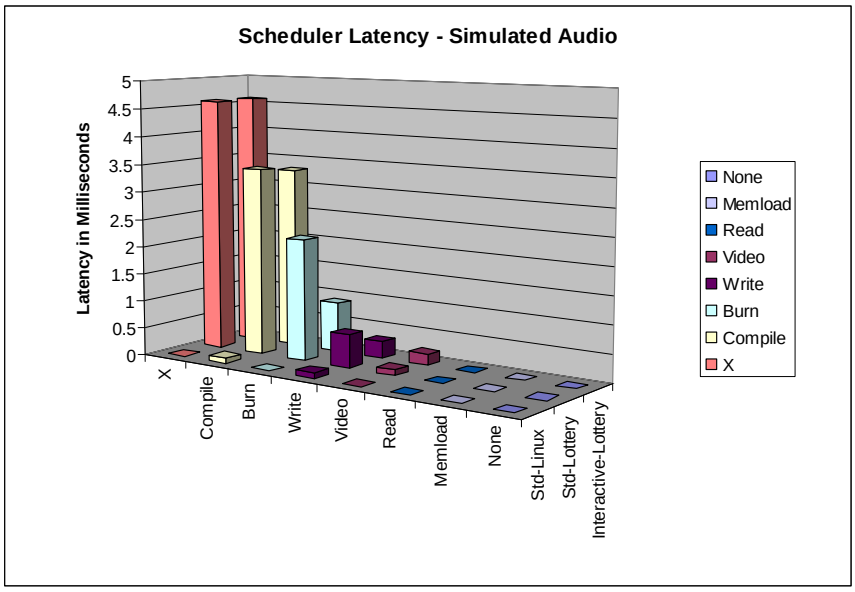

Figure 42 Simulated Audio, the most "interactive" of all the benchmarked processes, experiences relatively long latencies with either lottery scheduler in the presence of CPU bound additional load.

The results suggest that when the benchmark tasks are particularly "interactive" (e.g. Audio and Video), the introduction of CPU bound load has a very negative impact on the lottery scheduler's ability to service the benchmarked process. The more significant the CPU bound load the more impact on the interactive process. Only in cases where the benchmarked task uses the CPU heavily (e.g. Gaming) do the lottery scheduler's latencies compare favorably with the standard Linux kernel's scheduler. Both lottery schedulers favor CPU bound tasks to such a degree that it is clear that the compensation ticket function does not work as well as it was designed to work. Even the lottery scheduler that emphasizes interactive process support through distribution of extra compensation tickets does not service this class of processes as well as the standard Linux kernel's scheduler.

5.5 Kernel Compile Comparison

As an additional performance comparison a kernel compilation benchmark was performed. Kernel compilation is a standard benchmark that exercises both CPU and disk I/O. This compilation was performed in single user mode to limit the impact of other processes. The benchmark was performed using the standard Linux kernel, the standard lottery scheduling kernel, and the lottery scheduling kernel enhanced for "I/O bound/interactive" processes. A complete compilation was performed three times in each configuration and the average of the three is reported as the result. The benchmark was performed with no load, additional disk I/O load (disk reads and writes), additional user-mode CPU load, and additional kernel-mode CPU load.

The expectation was to see behavior consistent with the performance of the Interbench test; both lottery scheduling kernels tend to favor CPU bound processes. This means that compiles with no additional load will probably be similar between all kernels. Compiles with any type of I/O load will probably be faster with the lottery schedulers because the lottery scheduler will favor the CPU bound facet of the compile more than the standard Linux kernel. At the same time this means that the I/O load is not getting serviced as well, however since there is no measure of the additional loads' performance, this will not be seen. Compiles with any type of CPU bound load should be slower with either of the lottery schedulers than with the standard Linux kernel. There is not any expected difference between the kernel-mode and user-mode CPU load, but it is worth trying.

Here is how the compilations were performed. The system was first booted into init state 3 and then brought back to init state 1 (single user mode). If the test had any

additional load, it was a shell script started as a background job. Within a minute of starting any additional load a “make clean” was performed on the source tree and “time make bzImage” was launched from the command line. The “time” utility was used to report the execution time of the kernel compile. After the first compile completed, two more cycles of “make clean”, “time make bzImage” were performed. The “*real*” (wallclock) times of all compiles were recorded.

Below are the averaged results of the three compiles and a description of any additional load applied (Table 7).

Table 7 Summary results of the kernel compile benchmark, 3 run averages.

Average Kernel Compile Times (mm:ss)			
	Std-Linux	Std-Lottery	Interactive-Lottery
No Additional Load	4:04	4:05	4:03
With Disk I/O load	14:28	12:44	12:50
With User-Mode CPU load	7:29	11:33	10:46
With Kernel-Mode CPU load (*/bin/cat*)	10:13	4:09	4:10
With Kernel-Mode CPU load (*sh read*)	7:29	11:25	10:46

The results are consistent with the expectations as compiles on lottery scheduler kernels are faster in the presence of I/O load and slower in the presence of CPU load. The one exception to this is the test run with additional CPU load through the shell script using /bin/cat. In the detailed results discussion that follows this is explored in depth. By looking at the times of each separate compile, more details can be seen. In particular, the lottery schedulers’ preferential treatment of CPU bound processes is even more apparent. A brief analysis of the detailed results follows.

5.5.1 No Load

The detailed results of the compiles with no load (Table 8) show no difference in the performance of either of the lottery schedulers or of the standard Linux scheduler. The first compile is very slightly slower in all three cases, likely because the two latter compiles have the benefit of warmed disk caches.

Table 8 Detailed results from compile tests with no additional load show similar performance. Slower initial compiles in the first row of the table are likely from cold disk-cache.

Detailed Compile Times (mm:ss) No Additional Load		
Std-Linux	Std-Lottery	Interactive-Lottery
4:09	4:10	4:08
4:02	4:03	4:01
4:01	4:02	4:02

5.5.2 Disk I/O Load

In the results shown below, the compile with additional I/O load from disk activity shows significant variability (Table 9). The variation seen here is also related to cached disk data. First an explanation of exactly how the additional load was created. The following shell script was started as a background job. This performs a recursive listing of the entire file system starting at the root.

```
#!/bin/sh
while true; do ls –lR  /  > /tmp/list ; done  2>/dev/null
```

Table 9 Detailed results with additional I/O load. The lottery schedulers' bias against I/O activity is apparent, as the job of warming disk cache for the I/O load is penalized by both lottery scheduling kernels.

Detailed Compile Times (mm:ss) Disk I/O Load		
Std-Linux	Std-Lottery	Interactive-Lottery
18:09	4:31	4:34
12:38	16:49	16:40
12:38	16:53	17:16

Notice that the first compile with the standard Linux kernel took considerably longer than the next two. This extra time was due to disk caches being cold. After disk caches were loaded, the execution time stabilized on the second and third runs of the compile. In the case of the lottery scheduler, the results were reversed and reflected the lottery scheduler's CPU bound process bias. The initial compile (directly after the system boot and before the disk caches are hot) was much faster than subsequent compiles. This was because during the first compile the CPU bound compile processes were progressing much faster than the background disk I/O load. The second and third compiles took increasing amounts of time. By the third compile the execution time stabilized and while not shown in these results, subsequent compiles took a similar amount of time. The variability between the first compile and latter ones was explored more, and the effect of the disk caching could be seen using the iostat command. When the background load was initially started, iostat (reporting results at 5 second intervals) showed heavy disk read activity for over a minute as the first recursive read of the directory started. After the first read of the entire directory structure was complete (after 2 or 3 minutes), iostat showed that there was no longer

any read activity, only write. This was because all of the inode and related filesystem data was cached. Once all of the background load's data was cached, the CPU usage of the background load increased, and kernel compile times stabilized at about 16-17 minutes per compile.

5.5.3 User-Mode CPU Load

Compiling with additional user-mode CPU also resulted in varied results. The additional user mode CPU load was generated with the following shell script.

```
#!/bin/sh
while true; do (( 3+4 )) >/dev/null ; done
```

As seen in the results displayed below (Table 10), the relatively more I/O bound compile process took much more time to complete with either of the lottery schedulers. The standard Linux scheduler's compile times were more than 25% better when compared to either lottery scheduler. When comparing the two lottery schedulers to each other in the presence of CPU load, the relatively I/O bound compile did perform slightly better (about 7%) on the lottery scheduler tuned for interactive processes.

Table 10 Detailed results of the kernel compile benchmark with additional user-mode CPU bound load.

Detailed Compile Times (mm:ss) User-Mode CPU Load		
Std-Linux	Std-Lottery	Interactive-Lottery
7:42	12:31	11:42
7:22	11:09	10:20
7:25	11:01	10:16

5.5.4 Kernel-Mode CPU Load (/bin/cat)

The compile test was also performed with additional kernel-mode CPU load. The additional load was generated with the following shell script.

```
#!/bin/sh
while true; do cat /proc/interrupts > /dev/null; done
```

Continuous reads from /proc/interrupts generate a good amount of kernel mode activity. Note the use of cat (*/bin/cat*) to read from the /proc filesystem. The results of this test (Table 11) were unexpected.

Table 11 Using */bin/cat* to generate additional CPU load resulted in faster compile times for lottery schedulers; pointing to the lottery scheduler's inability to spawn processes as quickly as the standard Linux kernel.

Detailed Compile Times (mm:ss) Kernel-Mode CPU Load (*/bin/cat*)		
Std-Linux	Std-Lottery	Interactive-Lottery
10:13	4:15	4:14
10:13	4:07	4:08
10:14	4:07	4:09

Interestingly the lottery scheduling compile times were much shorter than the standard Linux scheduler times. While the compile completed much faster under the lottery schedulers, the additional kernel-mode CPU load progressed much slower than on the standard Linux scheduler. The standard Linux scheduler allowed the additional load to progress much faster, albeit with slower compile times. With an

alteration to the shell script the iterations through the while loop were counted. In the time it took to compile under the standard lottery scheduler roughly 44,000 iterations of the load loop were executed. In the time it took to compile with the standard Linux scheduler, over 600,000 iterations of the load were executed. There was an apparent imbalance in the rate at which the lottery schedulers were able spawn new processes as compared to the standard Linux scheduler. The lottery schedulers finished the compile times in not much longer than with no additional load. After further investigation it was determined that the shell script's spawning of the */bin/cat* process was the difference. To prove that the act of spawning the cat process was the differentiator, the kernel-mode CPU load shell script was revised (as seen below) to read from */proc/interrupts* without spawning a separate process.

5.5.5 Kernel-Mode CPU Load (*sh read*)

This script used the *sh* built-in *read* command to read from */proc/interrupts*.

```
#!/bin/sh
while true; do
while read myline; do
:
done < /proc/interrupts
done
```

The results from this test show the expected poorer performance under lottery scheduling (Table 12). As in the results from the User-Mode CPU loaded test, the lottery kernel tuned for interactive processes did allow the compile to complete a bit faster than the standard lottery scheduler.

Table 12 Generating additional load with the *sh* built-in *read* command results in performance similar to the benchmark with user-mode CPU load.

Detailed Compile Times (mm:ss) Kernel-Mode CPU Load (*sh read*)		
Std-Linux	Std-Lottery	Interactive-Lottery
7:40	12:09	11:39
7:25	11:04	10:17
7:24	11:04	10:24

5.6 Serving Web Pages with Apache

The purpose of this experiment is to measure the degree to which the performance of multiple web servers executing simultaneously on a single Linux server can be controlled with lottery ticket allocation as opposed to with the standard Linux *nice* function. The notion behind the test is that there may be a real-world application for a similar type of function; perhaps in a quality of service type of application where a guaranteed rate of operation is a requirement. The standard Linux kernel, the standard Lottery scheduling kernel, and the Lottery scheduling kernel enhanced for I/O bound processes were all tested as part of this experiment.

This experiment consisted of one Linux system running Apache web servers and three other Linux systems running the web server performance tool httperf. The four systems were networked together through a single 10/100Mb/s NetGear FS108 switch, and all operated at 100Mb/s. This benchmark was run with Apache 2.2.0, and httperf 0.8.

Since the focus of the test was to measure CPU resource allocation, basic optimizations were made with the intent of allowing the web server operate enough to load the CPU. For example if the web server performance failed because the network

stack failed before the CPU was loaded, the measure of control over the CPU resource is not consequential.

Httperf was compiled with a FD_SETSIZE of 65535 rather than the default of 1024. This was done to ensure that the httperf client did not max out the allowable number of open files before the web server became saturated. As a further step to avoid client overload before web server saturation, the default number of available TCP ports specified in the kernel of the client machines was changed. The following line was added to /etc/sysctl.conf on each of the clients.

net.ipv4.ip_local_port_range = 16384 65535

Several kernel parameters were altered on the system hosting the web servers as well. The following lines were added to the /etc/sysctl.conf file on the web server system.

net.core.rmem_max = 8738000
net.core.wmem_max = 6553600
net.ipv4.tcp_rmem = 8192 873800 8738000
net.ipv4.tcp_wmem = 4096 655360 6553600
net.ipv4.tcp_max_syn_backlog = 8192

These changes were made to keep the networking stack from failing before the web servers were fully taxing the CPU. Even with these changes only during the test runs with the smallest file being requested from the web-server were the clients able to push the CPU to 100% busy.

More configuration changes could have been made to further optimize the server and httperf clients, but part of the intent of the test was also to see how well the web servers could be controlled in a relatively standard configuration. An example of an additional configuration change that could have been made was to the maximum allowable number of httpd child processes (default is 256). This configuration option

controls the maximum number of child httpd processes that are to be spawned from the parent process. Under heavy load, the parent httpd process will spawn additional httpd processes to service the additional connection requests. These spawned processes share the total number of base tickets allocated to the user on a lottery scheduling kernel. On the standard kernel the spawned processes inherit the *nice* value of their parent.

In the more extreme test cases, the Linux system running the web servers would spawn the full 256 additional processes per instance and web servers would report in their error logs that the MaxClient limit had been exceeded. Even if this limit had been increased it is doubtful that performance would have improved as the system tended to thrash even more as the total process count went up. It was observed that with the ever increasing requests coming from the httperf benchmark, once the httpd child process count started to go up it usually went over the max value before the run of the benchmark completed. Additionally, degradation in web server performance correlated with the increase in httpd child processes.

On the Linux system running the Apache web servers, a dedicated web server instance was started for each client system running an httperf client. Each web server instance was started from a unique user account. The use of separate user accounts made it easier to run each instance with different resource rights to the CPU as the user account is a well defined resource boundary within the lottery scheduler.

When testing the web servers running on a lottery scheduling kernel, one of the web servers was given 400 tickets and the other two web servers were given 200 tickets. The idea was that the one web server favored with 400 tickets would perform

at the same rate as the other two combined. With the same intention, when testing with the standard Linux kernel the favored instance was left at the default nice value of 0 while the other two web servers were given nice values of 7.

From each of the httperf clients, to their corresponding web server instances, the requests per second was slowly raised. Each request to the web server called for a file that the web server was to send back to the client. In order to see how network I/O impacts the test, the experiment was performed with a small file (44 bytes), a medium sized file (17973 bytes) and a larger sized file (35946 bytes). The throughput of the web servers and response times at the clients was recorded and analyzed.

The lottery scheduler only showed any ability at all to control the level of web server performance in the case of the smallest sized file. In the case of the small file test the CPU would go to 100% busy during the part of the test of the where the web server performance starts to level off. The CPU never went to 100% busy with either the medium or large file tests. Likely this is because with the larger files the load was more I/O bound than CPU. With the smaller file, the I/O was able to be completed faster leaving the CPU more availability to establish new http connections.

Below are results of the small file test from the Standard Lottery scheduling kernel that show some degree of control over the performance of the web server instances (Figure 43, Figure 44 and Figure 45). The user bill was given 400 lottery tickets while users paul and fred were given only 200 tickets. The average response time (responses/second) is charted with the green line. The number of connections established per second is charted with the red line. This test specified 4 document

requests per connection, so before the web server became saturated; there were 4 responses containing the requested document for each connection request. As the web server became overloaded, it was not able to generate 4 responses for each connection, and in fact it was not even able to accommodate all connection requests. The number of desired requests is displayed on the x-axis of the graph. At about 2000 document requests per second even bill's web server that was favored with 400 lottery tickets began to drop connection requests and also was not able to make 4 replies for connections that are successful. Bill's web server did stabilize at about 600 connections per second and 1500 replies per second (Figure 43).

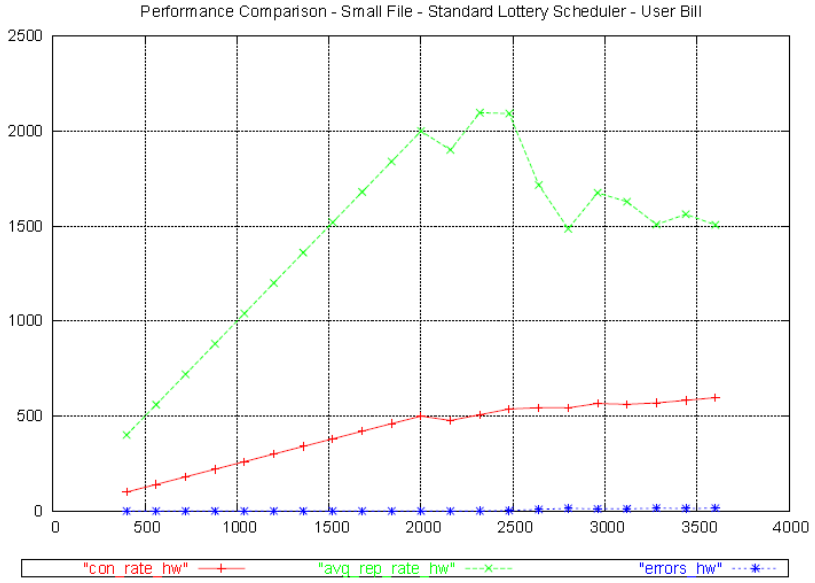

Figure 43 User bill's web server's connection rate, average reply rate, and error rate when transmitting a small file on the standard lottery scheduling kernel.

Fred and paul's web servers faltered earlier and stabilized at about 500 to 600 connections per second and 800 or so replies per second (Figure 44 and Figure 45). It

is likely that the spike in *average replies* (green line) at the end of the test is a side effect of bill's httperf session finishing slightly before fred and paul's. This early completion allowed fred and paul's web servers to finish strong.

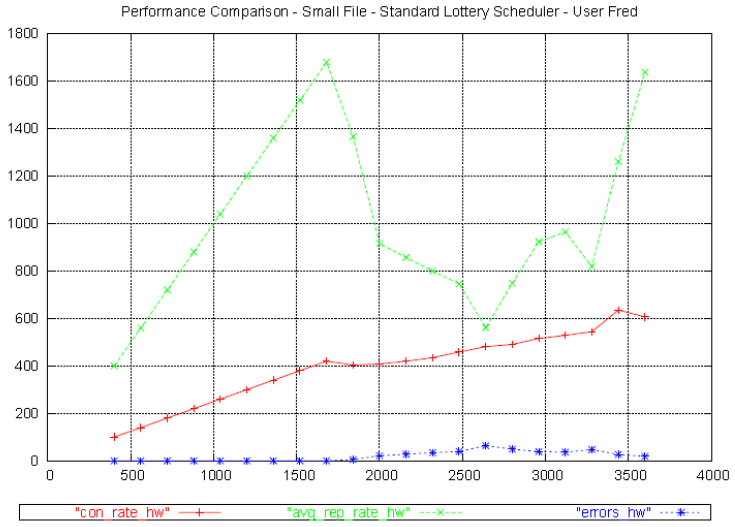

Figure 44 Small file test results for user fred who has half the ticket allocation of user bill.

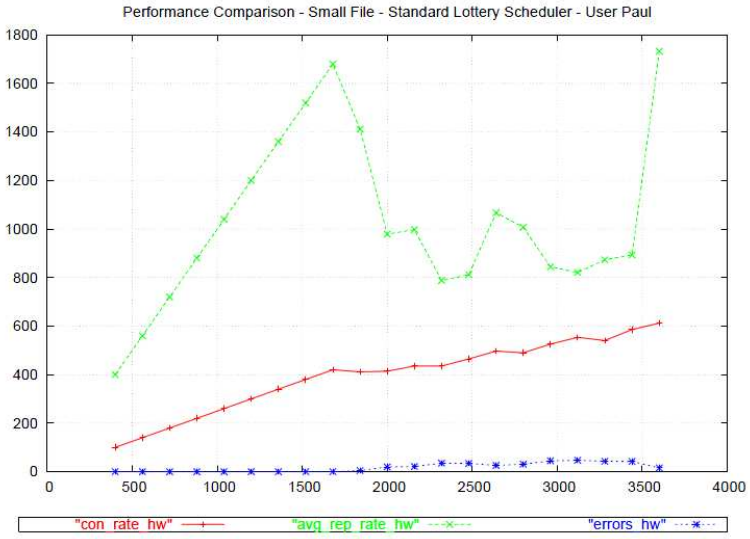

Figure 45 Small file test results for user paul who has half the ticket allocation of user bill.

The data from the small file test is consolidated to display side-by-side comparisons of actual reply rates (Figure 46). Multiple tests were performed between the Linux scheduler and the lottery schedulers and all displayed similar results. In Figure 46, the lottery scheduler shows an ability to control a relative rate of operation that is consistent with the ticket allocation ratio.

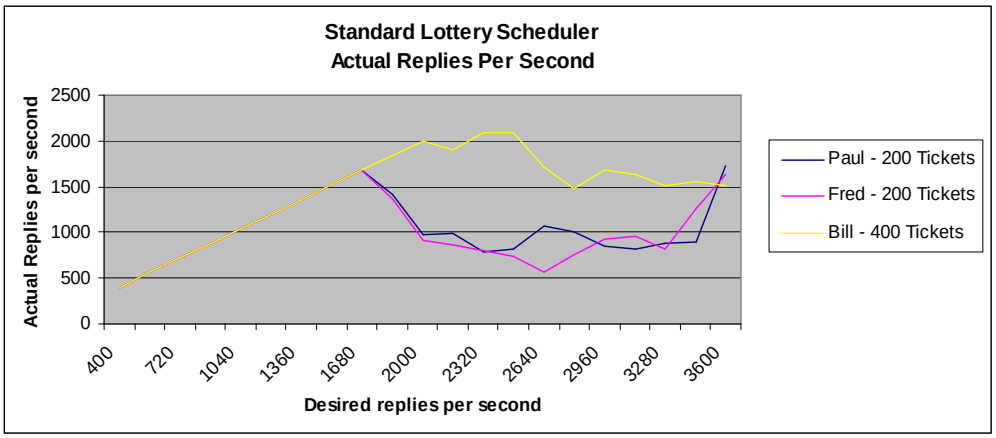

Figure 46 Small file response rates consolidated into a single graph. Some degree of control over the performance of the web servers is evident.

As seen in the graph below (Figure 47), all three web servers operate at roughly the same rate running under the standard Linux kernel's scheduler. In fact in this particular test run the web server with the most favorable "*nice*" level actually performs slightly worse. While no additional work was done to pursue the reason why, it would be interesting to investigate this unexpected result.

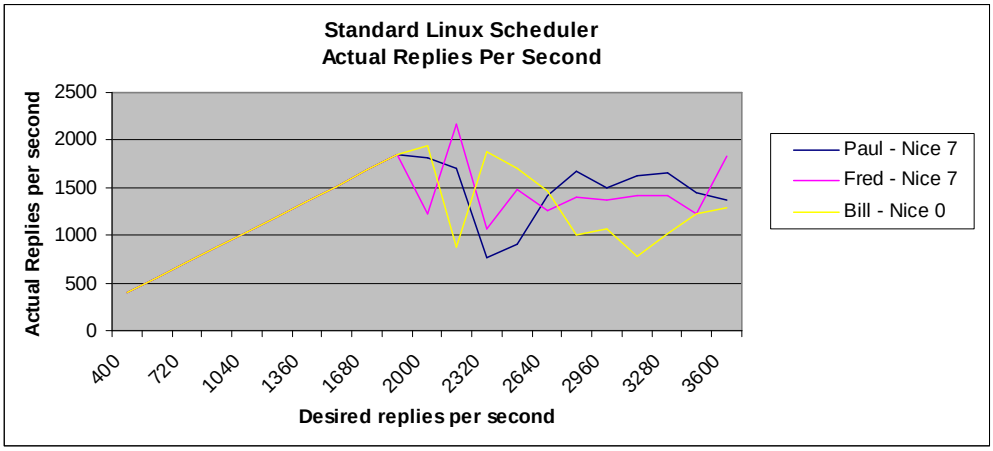

Figure 47 Small file test results with the standard Linux scheduler shows all web servers performing at a similar level.

The two graphs below (Figure 48 and Figure 49) display the rate of errors that each web server received. Again the results show that on the small file test the lottery scheduler was able to influence the relative performance of the web servers while the default Linux scheduler was not.

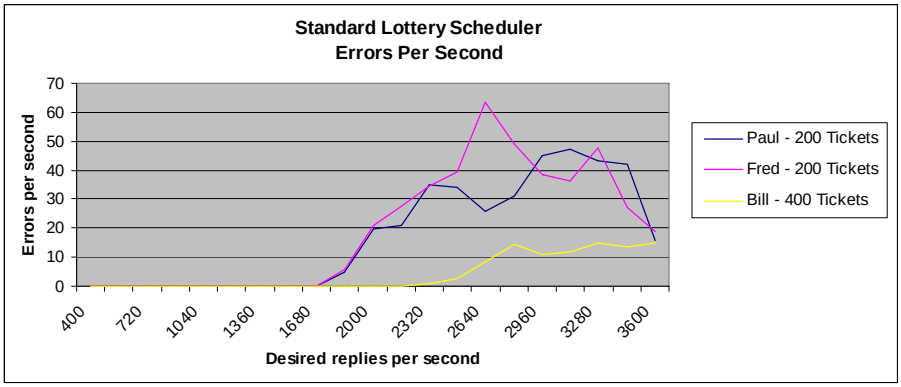

Figure 48 Small file test errors per second on the lottery scheduler correspond with the expected results. The web server with the greater number of lottery tickets has a lower error rate.

Just as the web server with the more favorable nice value performed slightly worse in Figure 47, the error rate for user bill as seen in Figure 49 is the opposite of what was expected. While bill's web server was running with a higher priority, the error rate was higher rather than lower.

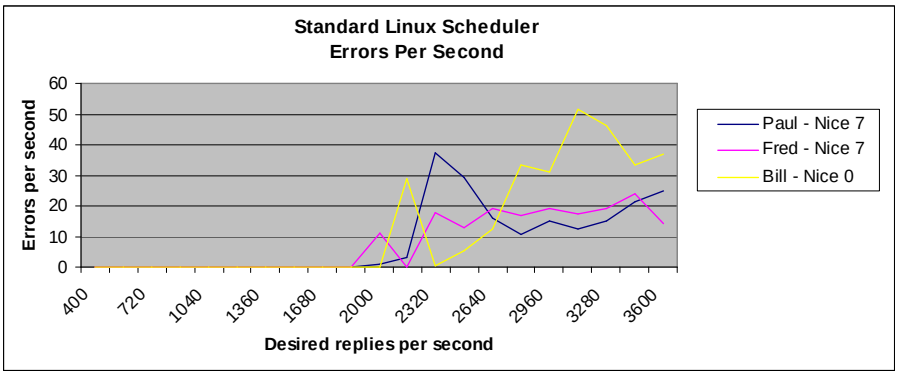

Figure 49 Small file test errors per second for the standard Linux scheduler did not show results consistent with the distribution of nice priorities. Higher priority led to higher error rate.

During the tests, network I/O peaked out at a little over 30 Mb/s on each httperf client on the medium and large file tests, while reaching only 2 or 3 Mb/s on the small file test. 30 Mb/s from each of 3 httperf clients is about 90 Mb/s for the server. Once this threshold was reached, performance from all running web servers suffered.

Test results for medium and large size files did not show any clear advantage for any of the schedulers with regards to controlling the relative rate of operation. While the lottery scheduler shows that it can control contention for the CPU, cases where the system bottle-neck is not the CPU are not managed any better by the lottery scheduler than the standard Linux scheduler.

6. Review of Current Works

Scheduling and computer resource allocation has long been a target of research. Several auction and market simulating resource allocation mechanisms have been proposed over the years. In “Incentive Engineering for Computational Resource Management”, Mark S. Miller and K. Eric Drexler (1988) propose a hybrid auction mechanism called the escalating-bid auction to be used for CPU allocation. Waldspurger et al. (Waldspurger, Hogg, Huberman, Kephart & Stornetta, 1992) implement a distributed computational economy (SPAWN) in which the CPU resource is the commodity of the economy. These microeconomic based resource allocation mechanisms propose that computer resource allocation be performed as the allocation of other resources are in a marketplace. While many of these economic-based approaches resolve the issue of fairness, they can be complex and carry corresponding overhead. Lottery scheduling is at the other end of the complexity spectrum. It too can "fairly" manage resources but in a simple fashion with minimal overhead.

Another scheduler that aims to be “fair” is the Share scheduler introduced by Kay and Lauder in their 1988 paper “A Fair Share Scheduler”. The Share scheduler’s goals are fairness between users, predictability, ease of use for end users, behavior that encourages load spreading, ensuring interactive users receive quality response, avoiding starvation, and dealing well with peak loads. The authors describe an algorithm that grants users shares on a system. These shares are analogous to the resource rights that lottery tickets grant to a user. The Share system decays these resource rights relative to the user’s usage of the system. In this way the Share

scheduler seeks to ensure fair-share allocation of the CPU resource. This is similar to lottery scheduling in that both schedulers seek to preserve users' resource rights.

The lottery scheduling idea was introduced by Waldspurger and Weihl (1994) in "Lottery Scheduling: Flexible Proportional-Share Resource Management". Each process is given a set of numbered lottery tickets. When the scheduler is ready to give the CPU to a process, it holds a "lottery", picks a random number, and the process with the winning ticket is given the CPU. Therefore a process's odds of receiving the CPU are directly related to the number of lottery tickets that have been allocated to the process. Though the concept can be used to allocate any type of resource the authors focus on the allocation of CPU cycles amongst competing processes. The paper differentiates the lottery method of scheduling from conventional priority based systems. An implementation in the Mach kernel was realized as proof of concept and is available as an artifact for further study. As the origin of the lottery method of resource management, this paper is an important read in order to understand the basis of the lottery concept and the other allocation strategies that evolved from it.

In "Proportional-Share Scheduling: Implementation and Evaluation in a Widely-Deployed Operating System", David Petrou and John Milford (1998) describe their experiences and findings in implementing lottery scheduling in the FreeBSD kernel. The authors describe the results of a straight implementation of the classic lottery model, and a second implementation that addresses some deficiencies that were identified in the first go round. The authors give a detailed analysis of the performance evaluation they performed on the scheduler. Their work also details

some good methods for obtaining measurement data, and some tests that are part of a meaningful evaluation. Petrou and Milford identify two main application classes to test, interactive and CPU bound. Petrou and Milford also identify a way to develop a good testing framework to examine scheduler performance against these two classes. The framework includes a method in which CPU cycles can be counted in different sections of the scheduler's code.

In a 1999 paper "Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers"; Petrou and Milford, along with Garth A. Gibson, follow up on their original work and implement additional extensions to the lottery scheduling algorithm. These extensions are in response to limitations discovered while implementing the classic algorithm. Many of these limitations manifest as poor response time for I/O bound processes. Their work focuses on the changes required to address these shortcomings, and the performance of their hybrid lottery scheduler. This implementation of lottery scheduling is also done in FreeBSD and the inspiration for the changes came from the standard FreeBSD scheduler which identifies and treats certain processes in a special manner.

Petrou, Milford and Gibson (1998, 1999) found subtle problems implementing lottery scheduling in its originally specified form. They found that while the lottery algorithm may provision the CPU fairly, overall system performance is not good. My implementation also deviates from traditional lottery scheduling theory, and also experienced poor response times for I/O bound processes. These two papers are extremely valuable to anyone who wishes to implement lottery scheduling; source code is available. The authors' descriptions of the engineering decisions required to

implement lottery scheduling provide a bridge from the theoretical aspects of the topic to the details of concept realization.

In 1995 Waldspurger and Weihl introduced another scheduler in their paper "Stride Scheduling: Deterministic Proportional-Share Resource Management". This scheduler too was designed to retain a high degree of control over proportional resource allocation like the lottery scheduler, but also sought to improve the variability in process latency, and to better control “throughput accuracy”. In the authors' comparison of the Stride scheduler to a lottery scheduler they identified the uneven progress that a process makes when being scheduled via the random nature of the lottery. They termed the measurement of this variability "throughput accuracy" and noted that the Stride scheduler's throughput accuracy was much more precise than that of the lottery scheduler. Also observed was that the Stride scheduler, when compared to a lottery scheduler, demonstrated a much lower degree of variability in a process' response time (latency); that is the time between a process being ready to run and actually running.

TFS or Time-Function Scheduling (Fong & Squillante, 1995) is similar to lottery scheduling in that it is designed to maintain a high degree of control over resource allocation. TFS also displays reduced variability in the waiting time that processes can experience in lottery scheduling. This is done with overhead comparable to that of lottery scheduling. TFS features dynamic priorities that change relative to the amount of time that a process has been waiting on a resource. In TFS processes are grouped together in classes that are defined by a time-function. Processes with similar characteristics and scheduling objectives are placed in the

same class. All job classes are divided into three job groups each with distinct priority. The highest priority grouping is reserved for system tasks. As suggested in the future works section of my paper, giving system tasks and the root user priority is a feature lacking in the lottery scheduler. In a comparison of TFS, lottery scheduling, and standard UNIX decay-usage policy scheduling, only TFS and lottery scheduling were able to achieve specific relative throughput objectives. Adjusting nice values allowed for some amount of control in the standard UNIX scheduler, but limited at best. These results are consistent with my findings. The TFS is also found to have a much lower level of variability in process wait time as compared to the lottery scheduler. The random nature of the lottery was noted to produce a high level of variability, and this is consistent with the variability that I identified in my work. Interestingly the authors also compared per class lottery scheduling to per job lottery scheduling. Lottery scheduling per class was performed by selecting a winning class through a lottery selection and then selecting a specific job from the winning class on a first-queued first-served basis. In this way the job in the winning class that had been sitting in the queue the longest was the winner. Intuitively lottery scheduling by class resulted in less variability in process latency.

In a 1999 paper "Tickets and Currencies Revisited: Extensions to Multi-Resource Lottery Scheduling" David Sullivan, Robert Hass, and Margo Seltzer describe a system where the lottery mechanism is used to control allocation of many resources including CPU time, memory, and IO bandwidth. Because different applications have different resource requirements, ticket exchange between applications is presented as a method to ensure effective resource distribution. The

exchange system proposed includes ticket negotiators working in the interest of applications and ticket brokers that facilitate inter-application ticket exchanges. Security concerns are addressed along with functional requirements. This paper is primarily theoretical and offers no practical implementation details. There are some comparisons made to Waldspurger and Weihl's (1994) original lottery resource allocation model, including one study of "nice" semantics that may be of interest to someone who is interested in implementing lottery scheduling.

Jennifer Spath's 1998 paper explores lottery scheduling in the Linux Kernel. The goal of Spath's thesis project was to implement lottery scheduling in the Linux kernel and compare system performance with the standard Linux scheduler. She also implemented compensation tickets to ensure that interactive processes were given equal CPU time. The paper gives a good review of the Linux scheduler circa 1998, and describes an implementation of lottery scheduling. Key sections of code written to implement the lottery scheduler are displayed and described. The comparison with the standard Linux scheduler is brief; the implementation of the scheduler seems to be the focus of the project. Spath provides an overview of the Linux scheduler in her paper. In this analysis she discusses some of the data structures that are used to support the functions of the scheduler. This presentation is useful because the Linux process structures are inter-twined with the scheduler structures, so the details are critical for an implementation. Spath also discusses her implementation of lottery scheduling, complete with code examples. She implements only a limited set of lottery scheduling mechanisms, a notable exclusion is currencies. This omission

restricts her scheduler from enforcing the fair-share characteristics of traditional lottery scheduling.

Hoque and Dey (2009) compare the abilities of lottery scheduling and the EEVDF (Stoica & Wahab, 1995) scheduler in scheduling real-time jobs. While the authors point out that lottery scheduling is not designed to guarantee a scheduling deadline, they test to see how well the lottery scheduler performs in this function. They alter the classic lottery scheduling algorithm by allocating an increasing number of lottery tickets to a real-time job as its deadline comes closer. EEVDF on the other hand is designed to ensure that deadlines are met or are only exceeded by a guaranteed limited amount of time. The authors find that if the real-time jobs have large periodic deadlines, the lottery scheduler performs comparably to the EEVDF scheduler.

7. Conclusion

Lottery scheduling is found to be very effective at dynamically controlling the relative rate of execution of multiple CPU bound processes. CPU bound process execution rates can be easily specified and dynamically altered with the scheduler's lottery ticket abstraction. Since lottery ticket allocations ensure access to the CPU in an amount proportional to the distribution, all users are ensured a fair share of the resource. Overall system performance is found to be lesser than that provided by the standard Linux kernel's scheduler. Specifically, the lottery scheduler's CPU bound process bias can unfairly penalize interactive loads. Interactive profile processes exhibit uneven performance when running in parallel with CPU bound loads. Certainly in my implementation, the classic compensation ticket function, as devised by Waldspurger and Weihl (1994), is not sufficient to meet the requirements of interactive loads. This finding regarding the classic compensation ticket function is consistent with the findings of Petrou, Milford, and Gibson (1999).

This research finds that the order in which processes win the lottery is not exactly in proportion to the distribution of lottery tickets in the short term. The random nature of the lottery results in an uneven pattern of wins and thus of CPU distribution when viewed over very short periods of time. In the lottery scheduler, the only notion of priority is the relative ratio of lottery ticket allotments. Thus it is possible for an interactive type process that may be deemed as high priority by another scheduler, to lose multiple consecutive lotteries while in possession of a superior ticket ratio. Over the short term this can result in uneven process execution.

In a practical application experiment where the lottery scheduler is used to control the service levels of simultaneously executing web servers; in only the rare cases where the web servers are CPU bound is the service level able to be affected.

The findings of this research suggest that while the lottery scheduler performs very well when managing CPU bound processes, I/O bound process are not serviced in a timely fashion and overall system performance for a general purpose system load is not as well balanced as it is under the standard Linux kernel.

8. Future Work

A primary area of interest for future work is exploring the I/O performance problem. The compensation ticket function does not work well in practice. Much of Petrou, Milford and Gibson's (1999) work focused on this problem. The I/O performance problem is a symptom of the overriding finding of my research; that lottery scheduling works great for managing CPU bound processes, but not necessarily for managing overall system performance. If some of these fundamental system performance problems could be overcome, there is other work that could be explored as part of using a lottery scheduler in a "live" general purpose system situation.

For example, one could investigate how an "under powered" root user impacts overall system performance on a heavily used multi-user system. For example, consider a situation where root user has significantly less tickets than a large group of users running multiple running CPU bound processes. Perhaps it would be necessary to "protect" the root account with some maximum/minimum ticket limits or ratios relative to total base tickets active on the system.

In my implementation, the only mechanism to change user base tickets is the *lsetbase* utility. A natural extension to this work would be a way to store a different starting base ticket level for each user. Then in the *alloc_uid* function where the *user_struct* for a user get allocated, the base ticket value could be set per user. This per-user default value could be stored in a /etc configuration file, e.g. */etc/passwd*.

As seen in the kernel compile test using /bin/cat to generate additional CPU load, the lottery schedulers spawned processes at a much slower rate. It would be

interesting to explore this and determine why. While likely an extension of the lottery scheduler's bias toward CPU bound process, ascertaining the details would be a worthwhile exercise.

Allowing 0 process and/or base tickets, as a mechanism for halting execution for a single process or for all processes owned by a single user, is another lottery scheduling extension that could be interesting to experiment with.

BIBLIOGRAPHY

Drexler, K. E., & Miller, M. S. (1988). Incentive Engineering for Computational Resource Management. In B. A. Huberman, (Ed.), The Ecology of Computation. (pp. 231-266). Amsterdam, Netherlands: North Holland.

Fong L. L., & Squillante M. S., (1995). Time-Function Scheduling: A General Approach to Controllable Resource Management. Yorktown Heights, NY: IBM Research Division, T. J. Watson Research Center.

Hoque E., and Dey T., (2009). Comparing Lottery and EEVDF Scheduling Algorithm for Real-time Applications. Charlottesville, VA: University of Virginia, Department of Computer Science School of Engineering and Applied Sciences.

Kay, J., & Lauder, P. (January 1988). A Fair Share Scheduler. Communications of the ACM, 31(1), 44-55.

Petrou, D., & Milford, J. (April 1998). Proportional-Share Scheduling: Implementation and Evaluation in a Widely-Deployed Operating System. Computer Science Department School of Computer Science, Carnegie Mellon University Selected Reports: Fall 1997 Software Systems Course, 17-28.

Petrou, D., Milford, J., & Gibson, G. (1999). Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. Proceedings of the 1999 USENIX Annual Technical Conference Monterey, California, USA, 1-14.

Spath, J. (1998). Lottery Scheduling in the Linux Kernel. Unpublished master's thesis, College of William and Mary, Williamsburg, Virginia.

Stoica I., & Abdel-Wahab H. (1995). Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation. (Technical Report TR-95-22) Norfolk, VA: Old Dominion University, Department of Computer Science.

Sullivan, D. G., Hass, R., & Seltzer, M. I. (1999). Tickets and Currencies Revisited: Extensions to Multi-Resource Lottery Scheduling. Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (Rio Rico, Arizona), IEEE Computer Society Press, 148-152.

Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1992). Numerical Recipes in C: The Art of Scientific Computing (2nd ed.). New York: Cambridge University Press

Waldspurger, C. A., Hogg, T., Huberman, B. A., Kephart, J. O., & Stornetta, W.S. (1992). Spawn: A Distributed Computational Economy. IEEE Transactions on Software Engineering 18(2), 103-117.

Waldspurger, C. A., & Weihl W. E. (1994). Lottery Scheduling: Flexible Proportional-Share Resource Management. Proceedings of the First USENIX Symposium on Operating System Design and Implementation, 1-11.

Waldspurger, C. A., Weihl, W. E., (1995). Stride Scheduling: Deterministic Proportional-Share Resource Management. (Technical Memorandum MIT/LCS/TM-528). Cambridge, MA: MIT Laboratory for Computer Science.